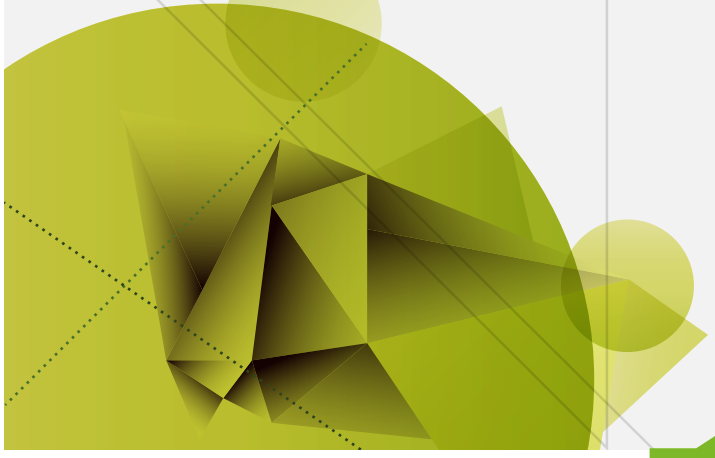
Bases de datos II



Autor: Julio Alberto Castillo

AREANDINA
Fundación Universitaria del Área Andina

MIEMBRO DE LA RED
ILUMNO

Bases de datos II / Julio Alberto Castillo, / Bogotá D.C.,
Fundación Universitaria del Área Andina. 2017

978-958-5460-29-4

Catalogación en la fuente Fundación Universitaria del Área Andina (Bogotá).

Edición:
Fondo editorial Areandino
Fundación Universitaria del Área Andina
Calle 71 11-14, Bogotá D.C., Colombia
Tel.: (57-1) 7 42 19 64 ext. 1228
E-mail: publicaciones@areandina.edu.co
http://www.areandina.edu.co

Primera edición: noviembre de 2017

Corrección de estilo, diagramación y edición: Dirección Nacional de Operaciones virtuales
Diseño y compilación electrónica: Dirección Nacional de Investigación

Hecho en Colombia
Made in Colombia

Bases de datos II

Autor: Julio Alberto Castillo

AREANDINA
Fundación Universitaria del Área Andina

MIEMBRO DE LA RED
ILUMNO

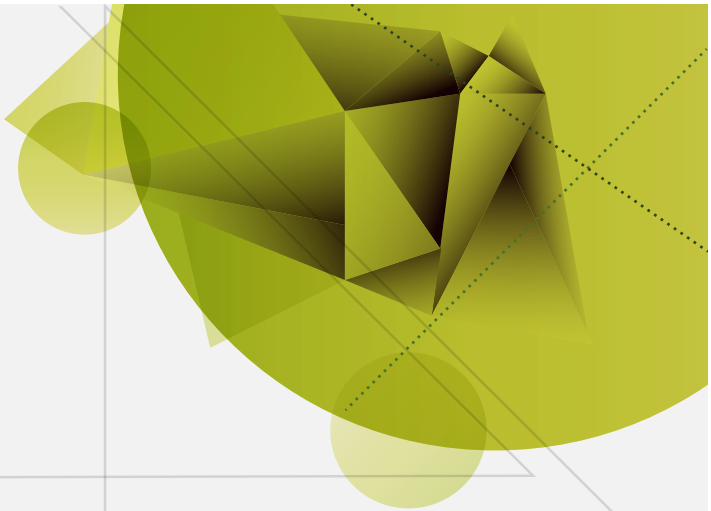
Índice

Índice

1

Unidad 1

Consultas avanzadas en MySQL



Bases de datos II

Autor: Julio Alberto Castillo

AREANDINA
Fundación Universitaria del Área Andina

MIEMBRO DE LA RED
ILUMNO

Introducción

Esta cartilla está dirigida a realizar un proceso de empalme y continuidad con respecto al módulo de Bases de datos II, teniendo en cuenta que en ella se ven conceptos básicos de esta temática, el propósito es profundizar y ampliar la cantidad de instrucciones, conceptos y métodos que los estudiantes pueden adquirir para administrar, diseñar y manejar Bases de datos relacionales.

Todo este proceso será realizado de manera incremental, teniendo como apoyo el desarrollo y aplicación de ejemplos que lleven al estudiante a entender y manejar de manera hábil dichas instrucciones, además que genera su propio estilo de manejo y administración, obviamente sin apartarse de los estándares y métodos que las diferentes entidades han implementado para su desarrollo.

Cada una de las instrucciones que explicaremos en este módulo será tendiente a resolver una problemática propuesta desde el inicio con el fin de llevar una secuencialidad en la forma de resolver las diferentes inquietudes y además generar coherencia dentro de los resultados que se pretenden obtener por parte de los estudiantes.

U1

Metodología

Con el fin de que el estudiante realice la mayor aprensión del conocimiento se realizaran las siguientes recomendaciones metodológicas:

Realizar las lecturas complementarias las cuales le permitirán ampliar conceptos y comprender la temática tratada en la unidad.

Utilizar fuentes bibliográficas e información de internet, recolectada para una mayor comprensión de la información sobre los temas propuestos.

Clasificar la información recolectada y realizar modelos aplicativos en los cuales el estudiante pueda corroborar y experimentar el funcionamiento y las diferentes maneras que hay para trabajar las múltiples funcionalidades que tiene el lenguaje SQL.

U1

Desarrollo temático

Consultas avanzadas en SQL

Tipos de Datos en MySQL

Al diseñar nuestras tablas tenemos que especificar el tipo de datos y tamaño que podrá almacenar cada campo. Una correcta elección debe procurar que la tabla no se quede corta en su capacidad, que destine un tamaño apropiado a la longitud de los datos, y la máxima velocidad de ejecución.

Básicamente MySQL admite dos tipos de datos: números y cadenas de caracteres. Junto a estos dos grandes grupos, se admiten otros tipos de datos especiales: formatos de fecha, etc.

Datos numéricos

En este tipo de campos solo pueden almacenarse números, positivos o negativos, enteros o decimales, en notación hexadecimal, científica o decimal. Los tipos numéricos tipo integer admiten los atributos SIGNED y UNSIGNED indicando en el primer caso que pueden tener valor negativo, y solo positivo en el segundo.

Los tipos numéricos pueden además usar el atributo ZEROFILL en cuyo caso los números se completaran hasta la máxima anchura disponible con ceros (column age INT (5) zerofill => valor 23 se almacenará como 00023).

BIT o BOOL: usado para un número entero que puede ser 0 o 1.

TINYINT: es un número entero con rango de valores válidos desde -128 a 127. Si se configura como unsigned (sin signo), el rango de valores es de 0 a 255.

SMALLINT: usado para números enteros, con rango desde -32768 a 32767. Si se configura como unsigned, 0 a 65535.

MEDIUMINT: para números enteros; el rango de valores va desde -8.388608 a 8388607. Si se configura como unsigned, 0 a 16777215

INT: para almacenar números enteros, en un rango de -2147463846 a 2147483647. Si configuramos este dato como unsigned, el rango es 0 a 4294967295

BIGINT: número entero con rango de valores desde -9223372036854775808 a 9223372036854775807. Unsigned, desde 0 a 18446744073709551615.

FLOAT (e,d): representa números decimales. Podemos especificar cuantos dígitos enteros (e) pueden usarse, y cuantos en la parte decimal (d). Mysql redondeará el decimal para ajustarse a la capacidad.

DOUBLE: número de coma flotante de precisión doble. Es un tipo de datos igual al anterior cuya única diferencia es el rango numérico que abarca

DECIMAL: almacena los números como cadenas.

Caracteres o cadenas

CHAR: este tipo se utiliza para almacenar cadenas de longitud fija. Su longitud abarca desde 1 a 255 caracteres.

VARCHAR: al igual que el anterior se utiliza para almacenar cadenas, en el mismo rango de 1-255 caracteres, pero en este caso, de longitud variable. Un campo CHAR ocupará siempre el máximo de longitud que le hallamos asignado, aunque el tamaño del dato sea menor (añadiendo espacios adicionales que sean precisos). Mientras que VARCHAR solo almacena la longitud del dato, permitiendo que el tamaño de la base de datos sea menor. Eso sí, el acceso a los datos CHAR es más rápido que VARCHAR.

No pueden alternarse columnas CHAR y VARCHAR en la misma tabla. Mysql cambiará las columnas CHAR a VARCHAR. También cambia automáticamente a CHAR si usamos VARCHAR con valor de 4 o menos.

TINYTEXT, TINYBLOB: para un máximo de 255 caracteres. La diferencia entre la familia de datatypes text y blob es que la primera se usa para cadenas de texto sin formato y que no distingue mayúsculas o minúsculas, mientras que blob se usa para objetos binarios: cualquier tipo de datos o información, desde un archivo de texto con todo su formato hasta imágenes, archivos de sonido y video

TEXT y BLOB: se usa para cadenas con un rango de 255 – 65535 caracteres. La diferencia entre ambos es que TEXT permite comparar dentro de su contenido sin distinguir mayúsculas y minúsculas, y BLOB si distingue.

MEDIUMTEXT, MEDIUMBLOB: puede almacenar textos de hasta 16777215 caracteres.

LONGTEXT, LONGBLOB: puede almacenar hasta máximo de 4.294.967.295 caracteres

Varios

DATE: para almacenar fechas. El formato por defecto es YYYY MM DD desde 0000 00 00 a 9999 12 31.

DATETIME: combinación de fecha y hora. El rango de valores va desde el 1 de enero del 1001 a las 0 horas, 0 minutos y 0 segundos al 31 de diciembre del 9999 a las 23 horas, 59 minutos y 59 segundos. El formato de almacenamiento es de año – mes - día horas: minutos: segundos.

TIMESTAMP: combinación de fecha y hora. El rango va desde el 1 de enero de 1970 al año 2037. El formato de almacenamiento depende del tamaño del campo

TIME: almacena una hora. El rango de horas va desde -838 horas, 59 minutos y 59 segundos a 838, 59 minutos y 59 segundos. El formato de almacenamiento es de 'HH:MM:SS.

YEAR: almacena un año. El rango de valores permitidos va desde el año 1901 al año 2155. El campo puede tener tamaño dos o tamaño 4 dependiendo de si queremos almacenar el año con dos o cuatro dígitos.

SET: un campo que puede contener ninguno, uno ó varios valores de una lista. La lista puede tener un máximo de 64 valores.

ENUM: es igual que SET, pero solo se puede almacenar uno de los valores de la lista.

Características de MySQL

El lenguaje de consulta estructurado (SQL) es un lenguaje de base de datos normalizado, utilizado por el motor de base de datos de MySQL. Para ejecutar un sinnúmero de acciones sobre los datos, los cuales son la esencia de este motor, es muy sencillo de manejar y tiene características similares a otros motores de Bases de Datos que incluso llegan a ser mucho más potentes pero que en general tienen la misma funcionalidad.

Componentes del SQL

El lenguaje SQL está compuesto por:

- Comandos.
- Cláusulas.
- Operadores.
- Funciones de agregado.

Estos elementos se combinan en múltiples instrucciones para crear, actualizar y manipular la información de la bases de datos.

Comandos

Existen dos tipos de comandos SQL:

- Los DLL que permiten crear y definir nuevas bases de datos, campos e índices.
- Los DML que permiten generar consultas para ordenar, filtrar y extraer datos de la base.

Comandos DLL	
Comando	**Descripción**
CREATE	Se usa para crear nuevas tablas, campos e índices.
DROP	Usado para eliminar tablas e índices.
ALTER	Usado para modificar las tablas agregando campos o cambiar la definición de los campos.

Cuadro 1
Fuente: Propia.

Comandos DML	
Comando	**Descripción**
SELECT	Consulta registros de la base que cumplan un criterio determinado.
INSERT	Utilizado para cargar grupos de datos en la base en una única operación o de forma individual.
UPDATE	Modifica los valores de los campos y registros especificados que cumplan una condición específica.
DELETE	Elimina registros de una tabla de una base que cumplan una condición específica.

Cuadro 2
Fuente: Propia.

Cláusulas

Las cláusulas son condiciones de modificación utilizadas como complemento para poder elegir los datos que desea manipular.

Cláusula	Descripción
FROM	Utilizada para especificar la tabla de la cual se van a seleccionar los registros.
WHERE	Especifica las condiciones que deben reunir los registros que se van a seleccionar.
GROUP BY	Separa los registros seleccionados en grupos específicos.
HAVING	Utilizada para expresar la condición que debe cumplir cada grupo.
ORDER BY	Ordena los registros de acuerdo a un orden específico.

Cuadro 3
Fuente: Propia.

Operadores lógicos

Operador	Uso
AND	Es el "y" lógico. Evalúa dos condiciones y devuelve un valor de verdad únicamente si ambas son ciertas.
OR	Es el "o" lógico. Evalúa dos condiciones y devuelve un valor de verdad si alguna de las dos es cierta.
NOT	Negación lógica. Devuelve el valor contrario de la expresión.

Cuadro 4
Fuente: Propia.

Operadores de comparación

Operador	Uso
<	Menor que.
>	Mayor que.
<>	Distinto de.
<=	Menor o Igual que.
>=	Mayor o Igual que.
=	Igual que.
BETWEEN	Utilizado para especificar un intervalo de valores Entre.
LIKE	Utilizado en la comparación de un modelo de dato Texto.
In	Utilizado para especificar registros de una base de datos.

Cuadro 5
Fuente: Propia.

Funciones de agregado

Las funciones de agregado se usan dentro de una cláusula SELECT en grupos de registros para devolver un único valor que se aplica a un grupo de registros.

Función	Descripción
AVG	Calcula el promedio de los valores de un campo determinado
COUNT	Usada para devolver el número de registros de la selección
SUM	Devuelve la suma de los valores de un campo determinado
MAX	Usada para devolver el valor más alto de un campo especificado
MIN	Usada para devolver el valor más pequeño de un campo especificado

Cuadro 6
Fuente: Propia.

Consultas de Selección

Las consultas de selección se utilizan para indicar al motor de datos que devuelva información de las bases de datos, esta información es devuelta en forma de conjunto de registros que se pueden almacenar en un registro. Este conjunto de registros es modificable.

Consultas básicas

La sintaxis básica de una consulta de selección es la siguiente:

SELECT Campos FROM Tabla;

En donde campos es la lista de campos que se deseen recuperar y tabla es el origen de los mismos, por ejemplo:

Imagen 1. Select
Fuente: Propia.

SELECT * FROM usuarios;

Esta consulta devuelve un registro con todos los campos de la tabla usuarios.

Ordenar los registros

Adicionalmente se puede especificar el orden en que se desean recuperar los registros de las tablas mediante la cláusula ORDER BY Lista de Campos. En donde Lista de campos representa los campos a ordenar. Ejemplo:

SELECT * FROM usuarios ORDER BY nombreusuario;

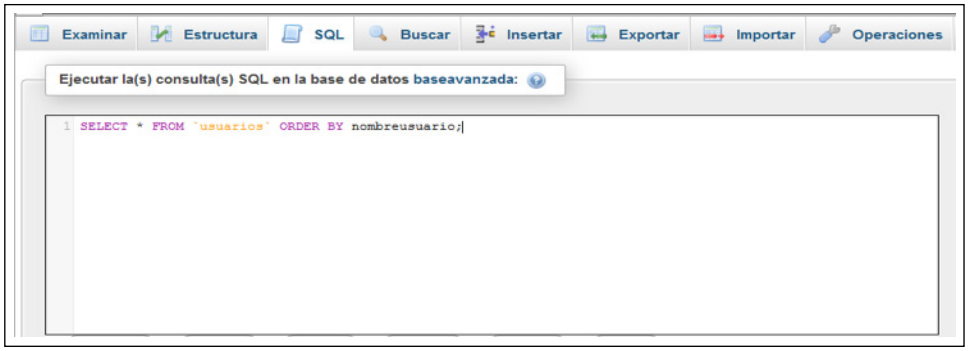


Imagen 2. Select
Fuente: Propia.

Esta consulta devuelve todos los campos de la tabla usuarios ordenados por el campo nombreusuario.

identificacion	nombreusuario	direccion	telefono	sexo	fechanac	tipousu	modalidad	codfacultad	clave
490787	ALEJANDRO PROSKAHUER MU?OZ		0	0	0000-00-00	2	1	258	aproskahuer
490787	ALEJANDRO PROSKAHUER MU?OZ		0	0	0000-00-00	1	1	258	aproskahuer
349969	ANGELINE KATHERINE VERGARA GRANDA		0	0	0000-00-00	1	1	210	anvergara4
492118	ANGIE JULIETH PAZ GONZALEZ		0	0	0000-00-00	1	1	357	apaz6
280716	ANTONIO LUIS ALBA BERDEAL		0	0	0000-00-00	1	1	185	aalba
280716	ANTONIO LUIS ALBA BERDEAL		0	0	0000-00-00	2	1	185	aalba
171761	BARBARA DE LAS MERCEDES MORA ESPINOZA		0	0	0000-00-00	2	1	210	bmora
171761	BARBARA DE LAS MERCEDES MORA ESPINOZA		0	0	0000-00-00	1	1	210	bmora
2474419	DIEGO ANDRES TORRES MONDRAGON		0	0	0000-00-00	1	1	257	dtorres29
2254216	GUSTAVO FIGUEROA LASSO		0	0	0000-00-00	1	1	114	gfigueroa3

Imagen 3. Resultado select
Fuente: Propia.

Se pueden ordenar los registros por más de un campo, como por ejemplo:

Imagen 4. Select y Order By
Fuente: Propia.

SELECT * FROM usuarios ORDER BY nombreusuario, identificacion;

Incluso se puede especificar el orden de los registros:

Ascendente mediante la cláusula (ASC este valor es por defecto).

Descendente (DESC).

SELECT * FROM `usuarios` ORDER BY nombreusuario, identificacion ASC

identificacion	nombreusuario	direccion	telefono	sexo	fechanac	tipousu	modalidad	codfacultad	clave
490787	ALEJANDRO PROSKAHUER MU?OZ		0	0	0000-00-00	2	1	258	aproskahuer
490787	ALEJANDRO PROSKAHUER MU?OZ		0	0	0000-00-00	1	1	258	aproskahuer
349969	ANGELINE KATHERINE VERGARA GRANDA		0	0	0000-00-00	1	1	210	anvergara4
492118	ANGIE JULIETH PAZ GONZALEZ		0	0	0000-00-00	1	1	357	apaz6
280716	ANTONIO LUIS ALBA BERDEAL		0	0	0000-00-00	1	1	185	aalba
280716	ANTONIO LUIS ALBA BERDEAL		0	0	0000-00-00	2	1	185	aalba
171761	BARBARA DE LAS MERCEDES MORA ESPINOZA		0	0	0000-00-00	2	1	210	bmora
171761	BARBARA DE LAS MERCEDES MORA ESPINOZA		0	0	0000-00-00	1	1	210	bmora

Imagen 5. Resultado Select Order
Fuente: Propia.

Consultas con predicado

El predicado se incluye entre la cláusula y el primer nombre del campo a recuperar, los posibles predicados son:

Predicado	Descripción
ALL	Devuelve todos los campos de la tabla
TOP	Devuelve un determinado número de registros de la tabla
DISTINCT	Omite los registros cuyos campos seleccionados coincidan totalmente
DISTINCTROW	Omite los registros duplicados basándose en la totalidad del registro y no sólo en los campos seleccionados.

Cuadro 7
Fuente: Propia.

ALL

Si no se incluye ninguno de los predicados se asume ALL. El Motor de base de datos selecciona todos los registros que cumplen las condiciones de la instrucción SQL. No es conveniente ejecutar demasiado este predicado ya que obligamos al motor de la base de datos a analizar toda la estructura de la tabla para verificar los campos que tiene, es mucho más rápido indicar el listado de campos deseados.

```
SELECT ALL FROM usuarios;
SELECT * FROM usuarios;
```

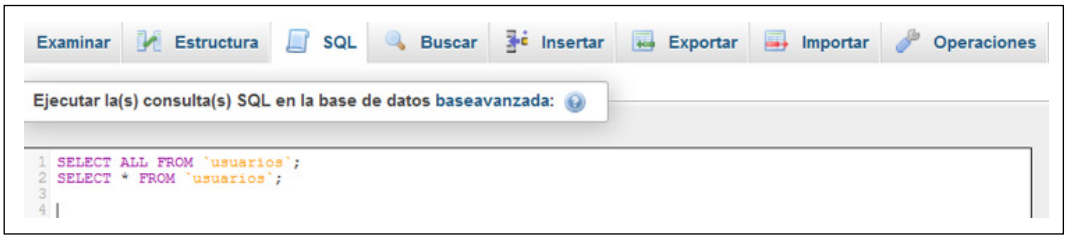


Imagen 6. Variaciones Select.
Fuente: Propia.

LIMIT

Devuelve un cierto número de registros delimitados específicamente por una cláusula ORDER BY. Supongamos que queremos recuperar los nombres e identificaciones de los 15 primeros usuarios:

```
SELECT nombreusuario, identificacion FROM usuarios ORDER BY nombreusuario DESC
LIMIT 15
```

nombreusuario	identificacion
YUSSEF KHAN	390402
YESID AROCA MORA	255958
YESID AROCA MORA	255958
VICTOR ALFONSO HERRERA CASTRO	2951274
VALENTINA TIJONOVA	267829
TOMAS EDUARDO RAMONES CORONA	333711
SILVIA JEANNETTE CONEDERA AGUIRRE	430463
RODRIGO ANDRES AYALA ZU?IGA	2518182
PEPITO PEREZ	1203698
OSCAR JOFRE MATEU	432733
NADIA TESSY SANCA VALDEZ	249900
MIRIAM ELIZALDE MOYAO	287099
MARIELYS KAROLINA SANJUAN IBARRA	472329
MARIA ROSEMEIRE DAMASCENO	254446
MARIA ROSEMEIRE DAMASCENO	254446

Imagen 7. Resultado Select Limit
Fuente: Propia.

Si no se incluye la cláusula ORDER BY, la consulta devolverá un conjunto arbitrario de 25 registros de la tabla usuarios .El predicado LIMIT no elige entre valores iguales.

DISTINCT

Omite los registros que contienen datos duplicados en los campos seleccionados. Para que los valores de cada campo listado en la instrucción SELECT se incluyan en la consulta deben ser únicos.

Por ejemplo, varios empleados listados en la tabla Empleados pueden tener el mismo apellido. Si dos registros contienen López en el campo Apellido, la siguiente instrucción SQL devuelve un único registro:

```
SELECT DISTINCT identificacion FROM usuarios;
```

Con otras palabras el predicado DISTINCT devuelve aquellos registros cuyos campos indicados en la cláusula SELECT posean un contenido diferente.

DISTINCTROW

Devuelve los registros diferentes de una tabla; a diferencia del predicado anterior que sólo se fijaba en el contenido de los campos seleccionados, éste lo hace en el contenido del registro completo independientemente del campo indicado en la cláusula SELECT.

```
SELECT distinctrow nombreusuario FROM `usuarios`
```

Si la tabla empleados contiene dos registros: Antonio Rodríguez y Marta Rodríguez el ejemplo del predicado DISTINCT devuelve un único registro con el valor Rodríguez en el campo nombreusuario ya que busca no duplicados en dicho campo. Este último ejemplo devuelve dos registros con el valor Rodríguez en el nombreusuario ya que se buscan no duplicados en el registro completo.

ALIAS

En determinadas circunstancias es necesario asignar un nombre a alguna columna determinada de un conjunto devuelto, otras veces por simple capricho o por otras circunstancias. Para resolver todas ellas tenemos la palabra reservada AS que se encarga de asignar el nombre que deseamos a la columna deseada.

Tomando como referencia el ejemplo anterior podemos hacer que la columna devuelta por la consulta, en lugar de llamarse nombreusuario (igual que el campo devuelto) se llame nombre. En este caso procederíamos de la siguiente forma:

```
SELECT distinctrow nombreusuario as nombre FROM usuarios
```

Criterios de selección

En la parte anterior hemos visto la forma de recuperar los registros de las tablas, las formas empleadas devolvían todos los registros de la tabla. Ahora tendremos la posibilidad de filtrar registros con el fin de recuperar únicamente aquellos registros que cumplan unas condiciones establecidas.

Antes de comenzar hay que tener en cuenta un detalle de mucha importancia. Cada vez que se desee establecer una condición referida a un campo de texto la condición de búsqueda debe ir encerrada entre comillas simples.

Operadores lógicos

Los operadores lógicos soportados por SQL son: AND, OR, XOR, Eqv, Imp, Is y Not. A excepción de los dos últimos todos poseen la siguiente sintaxis:

<expresión1> operador <expresión2>

En donde expresión1 y expresión2 son las condiciones a evaluar, el resultado de la operación varía en función del operador lógico. La tabla adjunta muestra los diferentes posibles resultados:

<expresión1>	Operador	<expresión2>	Resultado
Verdadero	AND	Falso	Falso
Verdadero	AND	Verdadero	Verdadero
Falso	AND	Verdadero	Falso
Falso	AND	Falso	Falso
Verdadero	OR	Falso	Verdadero
Verdadero	OR	Verdadero	Verdadero
Falso	OR	Verdadero	Verdadero
Falso	OR	Falso	Falso

Cuadro 8
Fuente: Propia.

Si a cualquiera de las anteriores condiciones le anteponemos el operador NOT el resultado de la operación será el contrario al devuelto sin el operador NOT.

SELECT * FROM usuarios WHERE codfacultad > 100 AND codfacultad < 150
SELECT * FROM usuarios WHERE (codfacultad > 100 AND codfacultad < 150) OR tipousu = 1; Intervalos de valores

Para indicar que deseamos recuperar los registros según el intervalo de valores de un campo emplearemos el operador Between cuya sintaxis es: campo [Not] Between valor1 And valor2 (la condición Not es opcional).

En este caso la consulta devolvería los registros que contengan en "campo" un valor incluido en el intervalo valor1, valor2 (ambos inclusive). Si anteponemos la condición Not devolverá aquellos valores no incluidos en el intervalo.

SELECT * FROM `usuarios` WHERE codfacultad between 100 and 200;
SELECT If(codfacultad Between 100 And 200, 'Facultad', 'Seccional') FROM usuarios

(Devuelve el valor Facultad si el código de la facultad se encuentra en el intervalo, 'Seccional' en caso contrario).

El Operador Like

Se utiliza para comparar una expresión de cadena con un modelo en una expresión SQL. Su sintaxis es:

Expresión Like modelo

En donde expresión es una cadena modelo o campo contra el que se compara expresión. Se puede utilizar el operador Like para encontrar valores en los campos que coincidan con el modelo especificado. Por modelo puede especificar un valor completo (Ana María), o se pueden utilizar caracteres comodín como los reconocidos por el sistema operativo para encontrar un rango de valores (Like An*).

El operador Like se puede utilizar en una expresión para comparar un valor de un campo con una expresión de cadena. Por ejemplo, si introduce Like C* en una consulta SQL, la consulta devuelve todos los valores de campo que comiencen por la letra C. En una consulta con parámetros, puede hacer que el usuario escriba el modelo que se va a utilizar.

El ejemplo siguiente devuelve los datos que comienzan con la letra P seguido de cualquier letra entre A y F y de tres dígitos:

Like 'P[A-F]###'

Este ejemplo devuelve los campos cuyo contenido empiece con una letra de la A a la D seguidas de cualquier cadena.

Like '[A-D]*' }

SELECT * FROM `usuarios` WHERE nombreusuario Like '[A-D]*'

En la tabla siguiente se muestra cómo utilizar el operador Like para comprobar expresiones con diferentes modelos.

Tipo de coincidencia	Modelo Planteado	Coincide	No coincide
Varios caracteres	'a*a'	'aa', 'aBa', 'aBBBa'	'aBC'
Carácter especial	'a[*]a'	'a*a'	'aaa'
Varios caracteres	'ab*'	'abcdefg', 'abc'	'cab', 'aab'
Un solo carácter	'a?a'	'aaa', 'a3a', 'aBa'	'aBBBa'
Un solo dígito	'a#a'	'a0a', 'a1a', 'a2a'	'aaa', 'a10a'
Rango de caracteres	'[a-z]'	'f', 'p', 'j'	'2', '&'
Fuera de un rango	'[!a-z]'	'9', '&', '%'	'b', 'a'
Distinto de un dígito	'[!0-9]'	'A', 'a', '&', '~'	'0', '1', '9'
Combinada	'a[!b-m]#'	'An9', 'az0', 'a99'	'abc', 'aj0'

Cuadro 9
Fuente: Propia.

El Operador In

Este operador devuelve aquellos registros cuyo campo indicado coincide con alguno de los en una lista. Su sintaxis es:

Expresión [Not] In(valor1, valor2, . . .)

SELECT * FROM usuarios WHERE nombreusuario In ('garzon', 'ramirez', 'castillo');

La cláusula WHERE

La cláusula WHERE es usada para determinar qué registros de las tablas enumeradas en la cláusula FROM aparecerán en los resultados de la instrucción SELECT. Después de escribir esta cláusula se deben especificar las condiciones. Si no se emplea esta cláusula, la consulta devolverá todas las filas de la tabla. WHERE es opcional, pero cuando aparece debe ir después del FROM.

SELECT nombreusuario, identificacion FROM usuarios WHERE codfacultad Between 100 And 300;

nombreusuario	identificacion
BARBARA DE LAS MERCEDES MORA ESPINOZA	171761
BARBARA DE LAS MERCEDES MORA ESPINOZA	171761
NADIA TESSY SANCA VALDEZ	249900
MARIA ROSEMEIRE DAMASCENO	254446
MARIA ROSEMEIRE DAMASCENO	254446
VALENTINA TIJONOVA	267829
LUCILA SIMONA SANCHEZ CALLE	273034
ANTONIO LUIS ALBA BERDEAL	280716
ANTONIO LUIS ALBA BERDEAL	280716
MIRIAM ELIZALDE MOYAO	287099
JUAN CARLOS RODRIGUEZ CARRERA	316515
JUAN CARLOS RODRIGUEZ CARRERA	316515
TOMAS EDUARDO RAMONES CORONA	333711
HECTOR OMAR MERCEDES HAZIM	347469
ANGELINE KATHERINE VERGARA GRANDA	349969

Imagen 8. Resultado Select Between
Fuente: Propia.

Agrupamiento de registros

GROUP BY

Combina los registros con valores idénticos, en la lista de campos especificados, en un único registro. Para cada registro se crea un valor sumario si se incluye una función SQL agregada, como por ejemplo Sum o Count, en la instrucción SELECT. Su sintaxis es:

SELECT campos FROM tabla WHERE criterio GROUP BY campos del grupo

GROUP BY es opcional. Los valores de resumen se omiten si no existe una función SQL agregada en la instrucción SELECT. Los valores Null en los campos GROUP BY se agrupan y no se omiten. No obstante, los valores Null no se evalúan en ninguna de las funciones SQL agregadas.

Se utiliza la cláusula WHERE para excluir aquellas filas que no desea agrupar, y la cláusula HAVING para filtrar los registros una vez agrupados.

SELECT codfacultad, Count(identificacion) FROM usuarios GROUP BY codfacultad;

Una vez que GROUP BY ha combinado los registros, HAVING muestra cualquier registro agrupado por la cláusula GROUP BY que satisfaga las condiciones de la cláusula HAVING.

HAVING es similar a WHERE, determina qué registros se seleccionan. Una vez que los registros se han agrupado utilizando GROUP BY, HAVING determina cuáles de ellos se van a mostrar.

SELECT codfacultad, Count(identificacion) FROM usuarios GROUP BY codfacultad Having Count(identificacion)>5;

Consultas de acción

Las consultas de acción son aquellas que no devuelven ningún registro, están encargadas de acciones como insertar y borrar y modificar registros.

DELETE

Elimina los registros de una o más de las tablas listadas en la cláusula FROM que cumplan con la cláusula WHERE. Este proceso elimina los registros completos, no es posible eliminar el contenido de algún campo en concreto. Su sintaxis es:

DELETE * FROM Tabla WHERE criterio

DELETE es especialmente útil cuando se desea eliminar varios registros. En una instrucción DELETE con múltiples tablas, debe incluir el nombre de tabla (Tabla.*). Si especifica más de una tabla desde la que eliminar registros. Si desea eliminar todos los registros de una tabla, eliminar la propia tabla es más eficiente que ejecutar borrado.

Una vez que se han eliminado los registros utilizando borrado, no puede deshacer la operación. Si desea saber qué registros se eliminarán, primero examine los resultados con una selección que utilice el mismo criterio y después ejecute el borrado. Mantenga copias de seguridad de sus datos en todo momento. Si elimina los registros equivocados podrá recuperarlos desde las copias de seguridad.

DELETE * FROM usuarios WHERE identificacion = 80472716;

Insert Into

Agrega un registro en una tabla. Insertar un único registro o Insertar en una tabla los registros contenidos en otra tabla.

Para insertar un único registró:

INSERT INTO Tabla (campo1, campo2, .., campoN) VALUES (valor1, valor2, ..., valorN) ;

Esta consulta graba en el campo1 el valor1, en el campo2 y valor2 y así sucesivamente. Hay que prestar especial atención a acotar entre comillas simples (') los valores literales (cadenas

de caracteres) y las fechas indicarlas en formato mm-dd-aa y entre caracteres de almohadillas (#).

Para insertar Registros de otra Tabla:

INSERT INTO Tabla (campo1, campo2, ..., campoN) SELECT TablaOrigen.campo1, TablaOrigen.campo2, ..., TablaOrigen.campoN FROM TablaOrigen

En este caso se seleccionarán los campos 1,2, ..., n de la tabla origen y se grabarán en los campos 1,2,.., n de la Tabla. La condición SELECT puede incluir la cláusula WHERE para filtrar los registros a copiar. Si Tabla y TablaOrigen poseen la misma estrucutra podemos simplificar la sintaxis a:

INSERT INTO Tabla SELECT TablaOrigen.* FROM TablaOrigen

Se puede utilizar la instrucción INSERT INTO para agregar un registro único a una tabla, utilizando la sintaxis correspondiente. Debe especificar cada uno de los campos del registro al que se le va a asignar un valor así como el valor para dicho campo. Cuando no se especifica dicho campo, se inserta el valor predeterminado o Null. Los registros se agregan al final de la tabla.

También se puede utilizar INSERT INTO para agregar un conjunto de registros pertenecientes a otra tabla o consulta utilizando la cláusula SELECT... FROM como se mostró anteriormente en la sintaxis de la consulta de adición de múltiples registros. En este caso la cláusula SELECT especifica los campos que se van a agregar en la tabla destino especificada.

La tabla destino u origen puede especificar una tabla o una consulta.

Si la tabla destino contiene una clave principal, hay que asegurarse que es única, y con valores no-Null; si no es así, no se agregarán los registros. Si se agregan registros a una tabla con un campo incremental, no se debe incluir el campo Contador en la instrucción.

INSERT INTO usuarios SELECT usuarios_otro.* FROM usuarios_Nuevos;
INSERT INTO usuarios (identificacion, nombreusuario, codfacultad) VALUES (12457896, 'Luis', 100);

UPDATE

Crea un proceso de actualización que cambia los valores de los campos de una tabla especificada basándose en un criterio específico. Su sintaxis es:

UPDATE Tabla SET Campo1=Valor1, Campo2=Valor2, ... CampoN=ValorN
WHERE Criterio;

UPDATE es especialmente útil cuando se desea cambiar un gran número de registros o cuando éstos se encuentran en múltiples tablas. Puede cambiar varios campos a la vez. El ejem-

plo siguiente incrementa los valores Cantidad pedidos en un 10 por ciento y los valores Transporte en un 3 por ciento para aquellos que se hayan enviado al Reino Unido.:

```
UPDATE Pedidos SET Pedido = Pedidos * 1.1, Transporte = Transporte * 1.03
WHERE PaisEnvío = 'ES';
```

UPDATE no genera ningún resultado. Para saber qué registros se van a cambiar, hay que examinar primero el resultado de una consulta de selección que utilice el mismo criterio y después ejecutar la consulta de actualización.

```
UPDATE Empleados SET Grado = 5 WHERE Grado = 2;
UPDATE Productos SET Precio = Precio  * 1.1 WHERE Proveedor = 8 AND Familia = 3;
```

Si en una consulta de actualización suprimimos la cláusula WHERE todos los registros de la tabla señalada serán actualizados.

```
UPDATE Empleados SET Salario = Salario * 1.1;
```

Consultas de Unión Internas

Las vinculaciones entre tablas se realizan mediante la cláusula INNER que combina registros de dos tablas siempre que haya concordancia de valores en un campo común. Su sintaxis es:

```
SELECT campos FROM tb1 INNER JOIN tb2 ON tb1.campo1 comp tb2.campo2;
```

En donde:

tb1, tb2 Son los nombres de las tablas desde las que se combinan los registros.

campo1, campo2 Son los nombres de los campos que se combinan. Si no son numéricos, los campos deben ser del mismo tipo de datos y contener el mismo tipo de datos, pero no tienen que tener el mismo nombre.

Se puede utilizar una operación INNER JOIN en cualquier cláusula FROM. Esto crea una combinación por equivalencia, conocida también como unión interna. Las combinaciones Equivalentes son las más comunes; éstas combinan los registros de dos tablas siempre que haya concordancia de valores en un campo común a ambas tablas. Se puede utilizar INNER JOIN con las tablas facultad y usuarios para seleccionar todos los usuarios de cada facultad. Por el contrario, para seleccionar todos los facultad (incluso si alguno de ellos no tiene ningún usuario asignado) se emplea LEFT JOIN o todos los usuarios (incluso si alguno no está asignado a ninguna facultad), en este caso RIGHT JOIN.

El ejemplo siguiente muestra cómo podría combinar las tablas facultad y usuarios basándose en el campo codfacultad:

```
SELECT nombreusuario, nomfacultad FROM facultad INNER JOIN usuarios ON facultad.
codfacultad = usuarios.codfacultad;
```

nombreusuario	nomfacultad
VICTOR ALFONSO HERRERA CASTRO	CONTADUR?A P?BLICA
JULIO ABSALON TORRES SUAREZ	MAESTR?A EN SALUD P?BLICA Y DESARROLLO SOCIAL
NADIA TESSY SANCA VALDEZ	ENFERMERIA
LUCILA SIMONA SANCHEZ CALLE	ENFERMERIA
LOUIS MONA	ENFERMERIA
GUSTAVO FIGUEROA LASSO	ESPECIALIZACION EN EPIDEMIOLOGIA
VALENTINA TIJONOVA	ESPECIALIZACION EN GERENCIA EN SALUD OCUPACIONAL

Imagen 9. Resultado Select Join
Fuente: Propia.

En el ejemplo anterior, codfacultad es el campo combinado, pero no está incluido en la salida de la consulta ya que no está incluido en la instrucción SELECT. Para incluir el campo combinado, incluir el nombre del campo en la instrucción SELECT, en este caso, facultad. codfacultad.

También se pueden enlazar varias cláusulas ON en una instrucción JOIN, utilizando la sintaxis siguiente:

```
SELECT campos FROM tabla1 INNER JOIN tabla2 ON tb1.campo1 comp tb2.campo1 AND
ON tb1.campo2 comp tb2.campo2) OR ON tb1.campo3 comp tb2.campo3)];
```

También puede anidar instrucciones JOIN utilizando la siguiente sintaxis:

```
SELECT campos FROM tb1 INNER JOIN (tb2 INNER JOIN [( ]tb3
[INNER JOIN [( ]tablax [INNER JOIN ...)] ON tb3.campo3 comp tbx.campox)] ON tb2.cam-
po2 comp tb3.campo3) ON tb1.campo1 comp tb2.campo2;
```

Un LEFT JOIN o un RIGHT JOIN puede anidarse dentro de un INNER JOIN, pero un INNER JOIN no puede anidarse dentro de un LEFT JOIN o un RIGHT JOIN.

Ejemplo

```
SELECT productos.codproducto, nomproducto, referencia, (valproducto * cantidad)
FROM productos INNER JOIN movimientos ON productos.codproducto = movimientos.
codproducto;
```

Crea dos combinaciones equivalentes: una entre las tablas movimientos y productos. Esto es necesario ya que la tabla moviientos no contiene datos de los productos. La consulta produce una lista de productos y sus ventas totales.

codproducto	nomproducto	referencia	(valproducto * cantidad)
1	Arroz	Libra	4000.00
1	Arroz	Libra	2400.00
2	Frijol	Libra	25000.00
2	Frijol	Libra	250000.00

Imagen 10. Resultado Select Inner Join
Fuente: Propia.

Si empleamos la cláusula INNER en la consulta se seleccionarán sólo aquellos registros de la tabla de la que hayamos escrito a la izquierda de INNER JOIN que contengan al menos un registro de la tabla que hayamos escrito a la derecha. Para solucionar esto tenemos dos cláusulas que sustituyen a la palabra clave INNER, estas cláusulas son LEFT y RIGHT. LEFT toma todos los registros de la tabla de la izquierda aunque no tengan ningún registro en la tabla de la izquierda. RIGHT realiza la misma operación pero al contrario, toma todos los registros de la tabla de la derecha aunque no tenga ningún registro en la tabla de la izquierda.

Consultas de unión externas

Se utiliza la operación UNION para crear una consulta de unión, combinando los resultados de dos o más consultas o tablas independientes. Su sintaxis es:

```
SELECT columna(s) FROM table1
UNION
SELECT columna(s) FROM table2;
```

Son instrucciones SELECT, el nombre de una consulta almacenada o el nombre de una tabla almacenada precedido por la palabra clave TABLE.

Puede combinar los resultados de dos o más consultas, tablas e instrucciones SELECT, en cualquier orden, en una única operación UNION. El ejemplo siguiente combina una tabla existente llamada Nuevas Cuentas y una instrucción SELECT:

```
SELECT nomcliente FROM clientes UNION SELECT nomproducto
FROM productos;
```

nomcliente
Julio Alberto Castillo
jorge Alberto Perez
Arroz
Frijol

Imagen 11. Resultado Select Union
Fuente: Propia.

Si no se indica lo contrario, no se devuelven registros duplicados cuando se utiliza la operación UNION, no obstante puede incluir el predicado ALL para asegurar que se devuelven todos los registros. Esto hace que la consulta se ejecute más rápidamente. Todas las consultas en una operación UNION deben pedir el mismo número de campos, no obstante los campos no tienen por qué tener el mismo tamaño o el mismo tipo de datos.

Se puede utilizar una cláusula GROUP BY y/o HAVING en cada argumento consulta para agrupar los datos devueltos. Puede utilizar una cláusula ORDER BY al final del último argumento consulta para visualizar los datos devueltos en un orden específico.

Consultas con parámetros

Las consultas con parámetros son aquellas cuyas condiciones de búsqueda se definen mediante parámetros. Si se ejecutan directamente desde la base de datos donde han sido definidas aparecerá un mensaje solicitando el valor de cada uno de los parámetros. Si deseamos ejecutarlas desde una aplicación hay que asignar primero el valor de los parámetros y después ejecutarlas. Su sintaxis es la siguiente:

```
PARAMETERS nombre1 tipo1, nombre2 tipo2, ... , nombreN tipoN Consulta
```

En donde:

nombre	Es el nombre asignado al parámetro.
tipo	Es el tipo de datos que tendrá el parámetro.
consulta	Una consulta SQL.

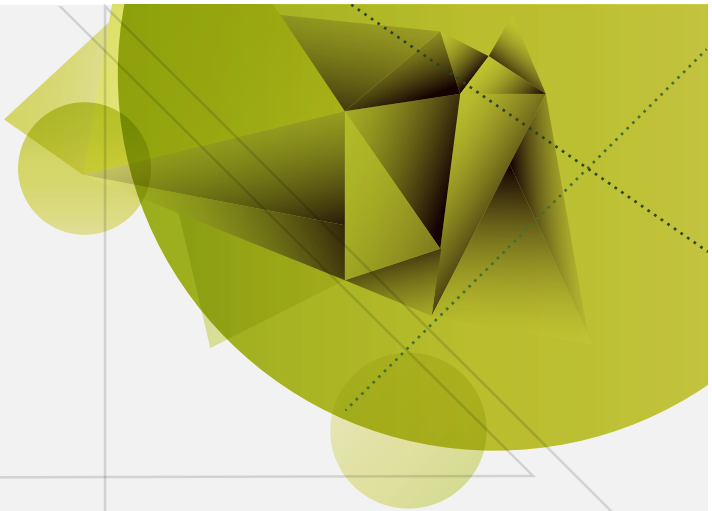
Se pueden utilizar nombres pero no tipos de datos en una cláusula WHERE o HAVING.

```
PARAMETERS
   PrecioMinimo Currency,
   FechaInicio DateTime;
SELECT
  IdPedido, Cantidad
FROM
   Pedidos
WHERE
   Precio = PrecioMinimo
   AND
   FechaPedido = FechaInicio
```

1

Unidad 1

Procesamiento de consultas



Bases de datos II

Autor: Julio Alberto Castillo

AREANDINA
Fundación Universitaria del Área Andina

MIEMBRO DE LA RED
ILUMNO

Introducción

El manejo de consultas es dentro de MySQL una de las características más importantes, este proceso nos permitirá realizar un uso adecuado de todas y cada una de las características que tienen estos elementos.

El procesamiento, manejo y desarrollo de las consultas le permitirá en gran manera hacer un buen trabajo de todos los recursos.

Analizar el esquema de la base de datos y ver si tiene sentido, es a menudo una de las tareas más complejas, las bases de datos tienen diseños malos y no están normalizadas. Esto puede afectar considerablemente la velocidad de las base de datos. Es por esto que esta semana está dirigida a conocer características especiales que logren un mejor desempeño de las consultas, así como los métodos y modelos que en conjunto con otras sentencias hagan de nuestra labor una experiencia interesante.

Cada una de las instrucciones que explicaremos en este módulo será tendiente a resolver una problemática propuesta desde el inicio con el fin de llevar una secuencialidad en la forma de resolver las diferentes inquietudes y además generar coherencia dentro de los resultados que se pretenden obtener por parte de los estudiantes.

U1

Metodología

Con el fin de que el estudiante realice la mayor aprensión del conocimiento se realizaran las siguientes recomendaciones metodológicas:

Realizar las lecturas complementarias las cuales le permitirán ampliar conceptos y comprender la temática tratada en la unidad.

Utilizar fuentes bibliográficas e información de internet, recolectada para una mayor comprensión de la información sobre los temas propuestos.

Clasificar la información recolectada y realizar modelos aplicativos en los cuales el estudiante pueda corroborar y experimentar el funcionamiento y las diferentes maneras que hay para trabajar las múltiples funcionalidades que tiene el lenguaje SQL.

U1 Desarrollo temático

Procesamiento de consultas

Qué son los índices en bases de datos

Imagina que tiene toda la información telefónica de los habitantes de un país como Colombia, con aproximada de 45 millones de habitantes no están ordenados.

Nota: para que los índices se noten realmente, debemos contar con un número significativo de registros en una tabla. La mejora suele ser proporcional y depende mucho de esa cantidad de registros.

Veamos una consulta como esta:

Imagen 1. Select general
Fuente: Propia.

Sin haber establecido orden en los datos, MySQL debe leer todos los registros de la tabla "personas" y realizar una comparación entre el campo "apellido" y la cadena de caracteres "rodriguez" para encontrar alguna coincidencia. A medida que esta base de datos tenga modificaciones, como un incremento en el número de registros, esta consulta requerirá el uso de memoria adicional necesaria para ejecutarse.

Si tuviéramos una guía telefónica localizaríamos fácilmente a cualquiera con apellido "rodriguez" yendo al final de la guía, a la letra "r". El método funciona por el ordenamiento de los datos y en el conocimiento de los mismos. En otras palabras, localizamos rápidamente a "rodriguez" porque está ordenado por apellido y porque nosotros conocemos el abecedario.

Si abrimos un libro veremos que posee un índice al inicio del libro, contenido por términos importantes con su correspondiente número de página. Si sabemos el tema buscamos la palabra y encontramos la expresión junto con su número de página.

Los índices de base de datos son muy similares. Al igual que el escritor decide crear un índice de términos y conceptos importantes de su libro, como administradores de una base de datos decidimos crear un índice respecto a uno o varios datos que tengan relevancia.

Nota: a lo largo de la búsqueda que realices sobre los índices en MySQL, se utilizan términos como INDICE o LLAVE para referirse a lo mismo.

Creando índices en MySQL

Usando el ejemplo inicial, para que la consulta anterior se ejecutase más rápido en nuestro sistema de datos, debemos crear un índice por apellido. Para crear ese índice podemos utilizar una sentencia en lenguaje SQL como la siguiente:

```
ALTER TABLE personas ADD INDEX (apellido)
```

De esta forma indicamos a MySQL que genere una lista ordenada de todos los registros pero ordenadas por los apellidos de la tabla personas, así como en el ejemplo del libro tenemos los números de teléfono ordenados por el apellido.

Nota: Es importante que cada una de estas instrucciones que vamos definiendo y conociendo en las cartillas de las diferentes semanas sean puestas en práctica en un servicio como PhpMyAdmin o en cualquier otro DBMS de su gusto.

Los índices se almacenan de manera que la base de datos pueda eliminar registros o filas determinadas por el resultado de una consulta ejecutada, estos datos son dinámicos y su gestión es transparente para el usuario o desarrollador.

Si no contamos con un método de Indexación, MySQL lee cada registro de forma secuencial, consumiendo demasiado tiempo, y utilizando muchas operaciones de ENTRADA/SALIDA en el disco e incluso llegando a dañar el sistema de caché del servidor.

Tampoco podemos llegar al otro extremo de crear un índice por cada columna de una tabla de MySQL, ya que esta necesita tener una lista separada de los valores de índice y actualizarlos conforme van cambiando. Al final, el manejo de índices requiere un equilibrio adecuado entre espacio de almacenamiento y tiempo de ejecución.

Es importante aclarar que una tabla con un campo indexado de MySQL usa más espacio y además tiene un BIT extra para realizar las consultas.

Optimizando consultas MySQL, creando y definiendo índices

Los índices se usan para buscar las filas con valores de columna específica rápidamente. Si no existe un índice, MySQL debe comenzar con el primer registro y luego realizar un recorrido a través de toda la tabla para buscar las demás filas que cumplan con el criterio o condición. Cuanto más grande sea la tabla, más demorado será este proceso. Si la tabla cuenta con un índice para las columnas que se están buscando, MySQL puede ubicar rápidamente la posición de búsqueda en el medio del archivo sin tener que mirar todos los datos. Si una tabla tiene 10.000 filas, entonces esto es por lo menos 100 veces más rápido que la lectura que se realiza de manera secuencial.

Todo esto conlleva que si una tabla tiene como Engine MyISAM puede dar lugar a bloqueos de registros mientras ésta se está consultando.

Los índices son elementos importantes y dándoles un buen uso permitirán el manejo óptimo de las tablas, pero si están mal diseñados o por el contrario no existen, la lentitud del sistema, el bloqueo de tablas e incluso el bloqueo y caída de un servidor se presenta, afectando el normal desarrollo del aplicativo que usa este motor como soporte de almacenamiento

Normalmente un índice se crea dependiendo de la búsqueda que se realice más frecuentemente, combinando campos que se encuentren en la sentencia SELECT y WHERE de la instrucción.

La sentencia SQL para crear un índice es la siguiente.

```
CREATE [UNIQUE|FULLTEXT|SPATIAL] INDEX index_name
[USING index_type] ON tbl_name (index_col_name,...) index_col_name:
col_name [(length)] [ASC | DESC]
```

Veamos ejemplos más específicos

Si seleccionamos la identificacion de usuario y el nombreusuario en una tabla usuarios buscando por el email, tendremos una instrucción SQL como la siguiente.

```
1 select identificacion,nombreusuario from usuarios  where llave = 'bmora';
```

Imagen 2. Select especifico
Fuente: Propia.

Ahora veamos que índices tiene creados. Esto se puede comprobar colocando delante de la SQL la sentencia EXPLAIN como se indica a continuación.

```
1 EXPLAIN SELECT identificacion, nombreusuario
2 FROM usuarios
3 WHERE llave = 'a*'
```

Imagen 3. Select explain
Fuente: Propia.

Esto nos muestra como resultado algo como lo que viene en la siguiente imagen.

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	usuarios	ALL	*NULL*	*NULL*	*NULL*	*NULL*	41	Using where

Imagen 4. Select resultado
Fuente: Propia.

Sql Explain

Para modificar un índice se puede hacer de la siguiente forma

Estableceremos que idlogins es el nombre del índice que estamos creando y nombreusuario el campo en la tabla que vamos a indexar.

Para añadir un nuevo índice compuesto se puede hacer de la siguiente manera:

```
1 alter table usuarios add index 'idlogins'
2 ('identificacion', 'nombreusuario', 'tipousu', 'llave');
```

Imagen 5. Alter table índice compuesto
Fuente: Propia.

Esto puede dar lugar a veces a que el tamaño del índice que estamos generando es demasiado grande.

Para que esto no nos ocurra, vamos ajustando el tamaño que asignamos a cada campo para el índice como lo indicamos a continuación.

```
1 ALTER TABLE usuarios ADD INDEX 'idlogin'
2 ('identificacion', 'nombreusuario' ( 200 ) , 'tipousu' , 'llave' ( 5 ) );
3
```

Imagen 6. Select alter add index.
Fuente: Propia.

Por esto **Explain** es una potente herramienta que MySQL que esta puesta a nuestra disposición para orientarnos sobre cómo está ejecutando una consulta en el motor de la base de datos.

Esto es de mucha utilidad cuando vamos a crear una consulta nueva, pues nos indicara que índices va a utilizar, de que tipo son, que tan efectiva será la respuesta al usar esos índices… por lo tanto, podemos detectar rápidamente, por ejemplo, que nuestra consulta no está bien estructurada o que nos falta un índice en algún campo especifico.

Si ejecutamos la siguiente consulta:

```
1 EXPLAIN SELECT *
2 FROM usuarios u
3 INNER JOIN facultad up ON u.codfacultad = up.codfacultad
```

Imagen 7. Select Explain Inner Join
Fuente: Propia.

Obtendremos una respuesta como esta:

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	up	ALL	PRIMARY,codfacultad	*NULL*	*NULL*	*NULL*	11	*NULL*
1	SIMPLE	u	ref	codfacultad	codfacultad	13	baseavanzada.up.codfacultad	1	*NULL*

Imagen 8. Resultado
Fuente: Propia.

Ahora veremos una breve explicación de los elementos que se utilizan para ajustar estas instrucciones y cuál es el concepto de cada uno de ellos.

1. Table: nos informa la tabla a la que nos estamos refiriendo. En el ejemplo anterior vemos los alias utilizados para tal propósito.
2. Type: el tipo de unión que se está usando en la generación de la consulta. Desde la mejor hasta la peor, los tipos de uniones son simple, const, eq_ref, ref, range, index, y ALL.
   - simple: tabla con una única fila, por tanto, la respuesta es inmediata y rápida.
   - Const: en la tabla coincide una única fila con los criterios indicados. Al sólo haber una sola fila como resultado, el optimizador toma este valor como una constante, por este motivo este tipo de tablas son muy rápidas.
   - Eq_ref: una fila de la tabla 1 será leída por cada combinación de filas de la tabla 2. Este tipo es usado cuando todas las partes de un índice se usan en la consulta y el índice es UNIQUE o PRIMARY KEY.
   - Ref: todas las filas con valores en el índice que coincidan serán leídos desde esta tabla por cada combinación de filas de las tablas previas. Similar a eq_ref, pero usado cuando usa sólo un prefijo más a la izquierda de la llave o si la llave no es UNIQUE o PRIMARY KEY. Si la llave que es usada coincide sólo con pocas filas, esta unión es buena.
   - Range: sólo serán recuperadas las filas que estén en un rango dado, usando un índice para seleccionar las filas. La columna key indica cual índice es usado, y el valor key_len contiene la parte más grande de la llave que fue usada. La columna ref será NULL para este tipo.
   - Index: barrido completo de la tabla para realizar cada combinación de filas de las tablas previas, revisando únicamente el índice.
   - ALL: recorrido completo de la tabla para cada combinación de filas. Es el peor caso ya que revisará todas las filas para cada combinación un proceso lento y consume demasiados recursos.
3. Possible_keys: posibles índices que utilizará la consulta.
4. Key: índice utilizado para ejecutar la consulta. Si indica el valor NULL, no se ha escogido ningún índice.
5. Key_len: cuanto más pequeño sea este valor, más rápida será la consulta, pues nos indica la longitud del índice usado.
6. Ref: las columnas del índice que se está usando, o una constante si esta es posible de acuerdo a la unión usada.
7. Rows: número de filas que MySQL debe analizar para devolver los datos solicitados.
8. Extra: información complementaria sobre como MySQL ejecutará la consulta. Los posibles valores en este campo pueden ser:
   - Distinct: MySQL ha encontrado una fila coincidente con los criterios indicados y no necesita seguir analizando.

- Not exists: MySQL fue capaz de hacer una optimización LEFT JOIN sobre la consulta y no examinará más filas en la tabla para la combinación de filas previa después de que encuentre una fila que coincida con el criterio LEFT JOIN.
- Range checked for each record: no se encontró ningún índice válido. Para cada combinación de filas se realizará un chequeo para poder determinar que índice utilizar y en caso de encontrar alguno válido, lo usará.
- Using filesort: este valor indica que MySQL necesita hacer un paso extra para encontrar la forma de ordenar las filas. Este tipo de consultas debe ser optimizada.
- Using index: recupera la información solicitada utilizando únicamente la información del índice. Esto sucede cuando todas las columnas requeridas forman parte del índice.
- Using temporary: para resolver esta consulta, MySQL creará una tabla temporal. Uno de los casos típicos en los que devuelve este valor es cuando usamos un ORDER BY sobre un conjunto de columnas diferentes a las indicadas en la cláusula GROUP BY. Este tipo de consultas debe ser optimizada.
- Where used: se usará una clausula WHERE para determinar que filas serán comparadas con otra tabla. Si no deseamos regresar todas las filas desde la tabla, y el Join es del tipo ALL o index, es muy probable que hayamos escrito algo mal en la consulta.

Si analizamos la consulta ejecutada anteriormente, vemos que para la tabla con alias 'up' está utilizando el tipo de unión 'ALL' debido a que no existe ningún filtro en la cláusula WHERE (en este caso, es normal ya que queremos devolver todos los valores). Se podría utilizar el índice existente pero al no haber ningún filtro determinado por el usuario, no es necesario usarlo. Para devolver esta consulta MySQL analizará 11 filas.

La segunda fila hace referencia a la tabla con alias 'u'. Utiliza un tipo de unión 'eq_ref' ya que devolverá una fila para cada fila de la tabla 'up' pudiendo usar y usando el índice primario de la tabla que tiene una longitud de 4 (muy pequeño). Se está utilizando la columna 'codfacultad' del índice de la tabla 'up' y de la base de datos 'baseavanzada' analizando una única fila para cada fila de la tabla referenciada.

Problemas al utilizar índices

También es cierto que no todo es bueno al utilizar índices. Está claro que las consultas de selección, si éstos están creados correctamente vuelan, pero las actualizaciones e inserciones se ven seriamente afectadas, por este motivo, sólo debemos crear índices sobre campos sobre los que estemos seguros que se van a realizar búsquedas a menudo, si no es el caso, es mejor no implementarlo.

La mejor forma de comprobar esto es haciendo uso de EXPLAIN, tanto para inserts (Inserciones) como para updates (Actualizaciones), de esta forma, podremos ver lo que necesita MySQL para llevar a cabo estos procesos.

Integridad referencial

Llaves foráneas en MySQL

Para que un campo sea una llave foránea, éste necesita ser definido como tal dentro del diseño y al momento de crear una tabla. Se pueden definir llaves foráneas en cualquier tipo de tabla de MySQL, pero únicamente tienen sentido cuando se usan tablas del tipo InnoDB.

A partir de las últimas versiones, se pueden definir restricciones de llaves foráneas con el uso de tablas InnoDB.

InnoDB es el primer tipo de tabla que permite definir estas restricciones para garantizar la integridad de los datos.

Para trabajar con llaves foráneas, necesitamos hacer lo siguiente:

Crear ambas tablas del tipo InnoDB.
Usar la sintaxis FOREIGN KEY(campo_fk) REFERENCES nombre_tabla (nombre_campo)
Crear un índice en el campo que ha sido declarado llave foránea.

InnoDB no crea de manera automática índices en las llaves foráneas o en las llaves referenciadas, así que debemos crearlos de manera explícita. Los índices son necesarios para que la verificación de las llaves foráneas sea más rápida. A continuación veremos cómo definir las dos tablas de ejemplo con una llave foránea.

```
1  CREATE TABLE clientes(
2      idcliente INT NOT NULL,
3      nombrecli VARCHAR(30),
4      direccion VARCHAR(80),
5      PRIMARY KEY (idcliente)
6  ) TYPE = INNODB;
7
8  CREATE TABLE ventas (
9      idfactura INT NOT NULL,
10     consecutivo INT NOT NULL,
11     idcliente INT NOT NULL,
12     cantidad   INT,
13     PRIMARY KEY(idfactura),
14     INDEX (idcliente),
15     FOREIGN KEY (idcliente) REFERENCES clientes(idcliente)
16 ) TYPE = INNODB;
17
```

Imagen 9 Create table Llaves Foraneas.
Fuente: Propia.

La sintaxis completa de una restricción de llave foránea es la siguiente:

```
[CONSTRAINT símbolo] FOREIGN KEY (nombre_columna, ...)
        REFERENCES nombre_tabla (nombre_columna, ...)
        [ON DELETE {CASCADE | SET NULL | NO ACTION
            | RESTRICT}]
        [ON UPDATE {CASCADE | SET NULL | NO ACTION
            | RESTRICT}]
```

Las columnas en la llave foránea y en la llave referenciada deben necesariamente tener tipos de datos similares para que puedan ser comparadas sin tener que hacer una conversión de tipos. El tamaño y el signo de los tipos enteros debe ser el mismo. En las columnas de tipo carácter (CHAR y VARCHAR), el tamaño de los campos no tiene que ser el mismo necesariamente.

Al evolucionar MySQL, se generan muchas mejoras dentro de las estructuras, por esto en las más recientes versiones se pueden agregar restricciones de llave foránea a una tabla con el uso de la sentencia ALTER TABLE. La sintaxis es:

```
ALTER TABLE nombre_tabla ADD [CONSTRAINT símbolo] FOREIGN KEY(...)
REFERENCES otra_tabla(...) [acciones_ON_DELETE][acciones_ON_UPDATE]
```

Por ejemplo, la creación de la llave foránea en la tabla venta que se mostró anteriormente pudo haberse hecho de la siguiente manera con el uso de una sentencia ALTER TABLE:

```
1  CREATE TABLE ventas
2  (
3      idfactura INT NOT NULL,
4      consecutivo INT NOT NULL,
5      idcliente INT NOT NULL,
6      cantidad   INT,
7      PRIMARY KEY(idfactura),
8      INDEX (idcliente),
9      FOREIGN KEY (idcliente) REFERENCES clientes(idcliente)
10 ) TYPE = INNODB;
11
12 ALTER TABLE ventas ADD FOREIGN KEY(idcliente) REFERENCES cliente(idcliente);
13
```

Imagen 10. Create table y Alter table
Fuente: Propia.

No debe usarse una sentencia ALTER TABLE en una tabla que está siendo referenciada, si se quiere modificar el esquema inicial de la tabla, se recomienda eliminar la tabla y volverla a crear con el nuevo esquema.

Cuando MySQL hace un ALTER TABLE, puede que use de manera interna un RENAME TABLE, y por lo tanto, se confundan las restricciones de llave foránea que se refieren a la tabla. Esta restricción aplica también en el caso de la sentencia CREATE INDEX, ya que MySQL la procesa como un ALTER TABLE.

Cuando se ejecute un script para cargar registros en una base de datos, es recomendable agregar las restricciones de llaves foráneas vía un ALTER TABLE. De esta manera no se tiene el problema de cargar los registros en las tablas de acuerdo a un orden lógico, este orden lógico está dado por la estructura y modelo creado de la base de datos, así que nosotros somos los llamados a determinar el orden en que se ejecutaran las instrucciones del script que estemos creando para que los registros ingresen de manera correcta a la base de datos y sean establecidas las restricciones correspondientes de manera óptima.

Integridad de dominio

La integridad de dominio viene dada por la validez de las entradas para una columna determinada. Puede exigir la integridad de dominio para restringir el tipo mediante tipos de datos, el formato mediante reglas y restricciones CHECK, o el intervalo de valores posibles mediante restricciones FOREIGN KEY, restricciones CHECK, definiciones DEFAULT, definiciones NOT NULL y reglas.

La regla de integridad de dominio establece dos condiciones.

La primera condición consiste en que un valor no nulo de un atributo Ai debe pertenecer al dominio del atributo Ai; es decir, debe pertenecer a dominio(Ai).

Esta condición implica que todos los valores no nulos que contiene la base de datos para un determinado atributo deben ser del dominio declarado para dicho atributo, esto suena un poco enrredado pero lo aclararemos con un.

Ejemplo:

Si en la relación EMPLEADOS(cedulaide, nombre, apellido, edademp) hemos declarado que dominio(cedulaide) es el dominio predefinido de los enteros, entonces no podremos insertar, por ejemplo, ningún empleado que tenga por cedulaide el valor “Luis”, que no es un entero.

Recordemos que los dominios pueden ser de dos tipos: predefinidos o definidos por el usuario. Los dominios definidos por el usuario resultan muy útiles, porque nos permiten determinar de forma más específica cuáles serán los valores admitidos por los atributos.

Ejemplo:

Supongamos ahora que en la relación EMPLEADOS(cedulaide, nombre, apellido, edademp) hemos declarado que dominio(edademp) es el dominio definido por el usuario y se llamara edad. Supongamos también que el dominio edad se ha definido como

el conjunto de los enteros que están entre 16 y 65. En este caso, por ejemplo, no será posible insertar un empleado que tenga una edad de 90 para el campo edademp.

La segunda condición de la regla de integridad de dominio es más compleja, especialmente en el caso de dominios definidos por el usuario; los SGBD actuales no la soportan para estos últimos dominios. Por estos motivos sólo la explicaremos de manera basica.

Esta segunda condición sirve para establecer que los operadores que pueden aplicarse sobre los valores dependen de los dominios de estos valores; es decir, un operador determinado sólo se puede aplicar sobre valores que tengan dominios que le sean adecuados.

Ejemplo:

Analizaremos esta segunda condición de la regla de integridad de dominio con un ejemplo concreto. Si en la relación EMPLEADOS(cedulaide, nombre, apellido, edademp) se ha declarado que dominio(cedulaide) es el dominio predefinido de los enteros, entonces no se permitirá consultar todos aquellos empleados cuyo cedulaide sea igual a 'Elena' (cedulaide = 'Elena'). El motivo es que no tiene sentido que el operador de comparación = se aplique entre un cedulaide que tiene por dominio los enteros, y el valor 'Elena', que es una serie de caracteres.

De este modo, el hecho de que los operadores que se pueden aplicar sobre los valores dependan del dominio de estos valores permite detectar errores que se podrían cometer cuando se consulta o se actualiza la base de datos. Los dominios definidos por el usuario son muy útiles, porque nos permitirán determinar de forma más específica cuáles serán los operadores que se podrán aplicar sobre los valores.

Ejemplo:

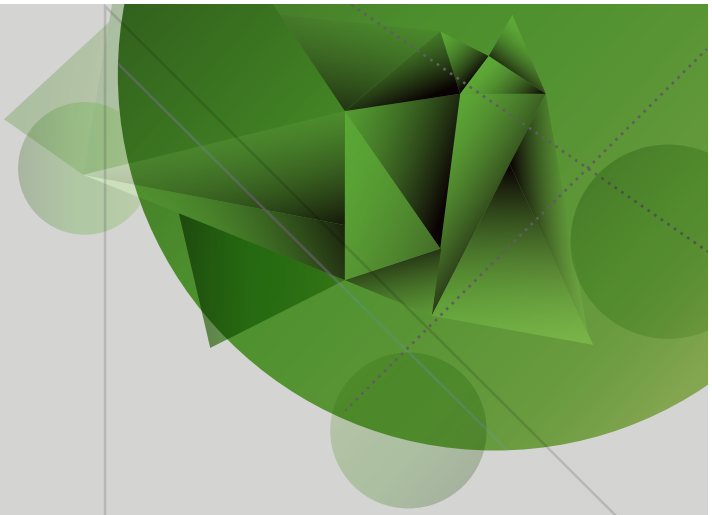
Veamos otro ejemplo con dominios definidos por el usuario. Supongamos que en la conocida relación EMPLEADOS (cedulaide, nombre, apellido, edademp) se ha declarado que dominio(cedulaide) es el dominio definido por el usuario números cedula y que dominio(edademp) es el dominio definido por el usuario edad. Supongamos que números cedula corresponde a los enteros positivos y que edad corresponde a los enteros que están entre 16 y 65. En este caso, será incorrecto, por ejemplo, consultar los empleados que tienen el valor de cedulaide igual al valor de edademp. El motivo es que, aunque tanto los valores de cedulaide como los de edademp sean enteros, sus dominios son diferentes; por ello, según el significado que el usuario les da, no tiene sentido compararlos.

Sin embargo para esta regla no existe un soporte suficiente por parte de los motores, así que si alguno de ustedes desea aplicarla debería establecer sus propias reglas y además diseñar una forma en que estas reglas de dominio de puedan ejecutar de manera explícita ya que los motores actuales no la soportan.

2

Unidad 2

Operadores y funciones



Bases de datos II

Autor: Julio Alberto Castillo

AREANDINA
Fundación Universitaria del Área Andina

MIEMBRO DE LA RED
ILUMNO

Introducción

Esta cartilla está dirigida a realizar un proceso de empalme y continuidad con respecto al módulo de Bases de Datos II, teniendo en cuenta que en ella se ven conceptos básicos de esta temática, el propósito es profundizar y ampliar la cantidad de instrucciones, conceptos y métodos que los estudiantes pueden adquirir para administrar, diseñar y manejar Bases de Datos relacionales.

Todo este proceso será realizado de manera incremental, teniendo como apoyo el desarrollo y aplicación de ejemplos que lleven al estudiante a entender y manejar de manera hábil dichas instrucciones, además que genera su propio estilo de manejo y administración, obviamente sin apartarse de los estándares y métodos que las diferentes entidades han implementado para su desarrollo.

Cada una de las instrucciones que explicaremos en este módulo será tendiente a resolver una problemática propuesta desde el inicio con el fin de llevar una secuencialidad en la forma de resolver las diferentes inquietudes y además generar coherencia dentro de los resultados que se pretenden obtener por parte de los estudiantes.

U2

Metodología

Con el fin de que el estudiante realice la mayor aprensión del conocimiento se realizaran las siguientes recomendaciones metodológicas:

Realizar las lecturas complementarias las cuales le permitirán ampliar conceptos y comprender la temática tratada en la unidad.

Utilizar fuentes bibliográficas e información de internet, recolectada para una mayor comprensión de la información sobre los temas propuestos.

Clasificar la información recolectada y realizar modelos aplicativos en los cuales el estudiante pueda corroborar y experimentar el funcionamiento y las diferentes maneras que hay para trabajar las múltiples funcionalidades que tiene el lenguaje SQL.

U2

Desarrollo temático

Operadores y funciones

Las expresiones pueden usarse en varios puntos de los comandos SQL, tales como en las cláusulas ORDER BY o HAVING de los comandos SELECT, en la cláusula WHERE de los comandos SELECT, DELETE, o UPDATE o en comandos, SET. Las expresiones pueden escribirse usando valores literales, valores de columnas, NULL, funciones y operadores.

En este capítulo se describen las funciones y operadores permitidos para escribir expresiones en MySQL.

Una expresión que contiene NULL siempre produce un valor NULL a no ser que se indique de otro modo en la documentación para una función u operador particular.

Nota: por defecto, no debe haber espacios en blanco entre un nombre de función y los paréntesis que lo siguen. Esto ayuda al motor de MySQL a distinguir entre llamadas a funciones y referencias a tablas o columnas que tengan el mismo nombre que una función. Sin embargo, se permiten espacios entre los argumentos de las funciones.

Puede decirle a MySQL server que acepte espacios tras los nombres de funciones arrancando con la opción --sql-mode=IGNORE_SPACE.

Los programas cliente pueden pedir este comportamiento usando la opción CLIENT_IGNORE_SPACE para mysql_real_connect().

En cualquier caso, todos los nombres de función son palabras reservadas. La totalidad de ejemplos de esta semana, serán ejecutados desde PHPMYADMIN y muestran la salida del programa MySQL:

Operadores

Al hablar de operador se hace referencia a un símbolo que especifica una acción que se realiza en una o más expresiones. Y sus categorías son:

- Operadores aritméticos

Sirven para realizar operaciones matemáticas entre dos o más expresiones numéricas.

Operador	Significado
+	Suma
-	Resta
*	Multiplicación
/	División
%	Módulo

- Operadores de asignación

Se utiliza para llevar información o un valor a una constante o variable.

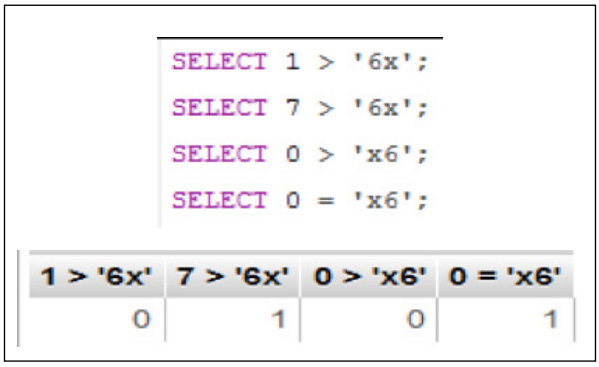
```
DECLARE @CANTIDAD INT
SET @CANTIDAD=5
```

Operador	Significado
=	Asignación

- Operadores de comparación

Facilitan la comprobación de dos expresiones, regresando un dato de tipo BOOLEAN. Y el valor retornado es TRUE o FALSE, las únicas expresiones que no soportan son lo image, text y ntext.

Operador	Significado
=	Igual a
>	Mayor que
<	Menor que
>=	Mayor o igual que
<=	Menor o igual que
<>	No es igual a
!=	No es igual a
!<	No es menor que
!>	No es mayor que



```
SELECT 1 > '6x';
SELECT 7 > '6x';
SELECT 0 > 'x6';
SELECT 0 = 'x6';
```

1 > '6x'	7 > '6x'	0 > 'x6'	0 = 'x6'
0	1	0	1

Imagen 1. Aplicación operadores
Fuente: Propia.

- = Igual:

```
SELECT 1 = 0;
SELECT '0' = 0;
SELECT '0.0' = 0;
SELECT '0.01' = 0;
SELECT '.01' = 0.01;
```

1 = 0	'0' = 0	'0.0' = 0	'0.01' = 0	'.01' = 0.01
0	1	1	0	1

Imagen 2. Aplicación Igual
Fuente: Propia.

- <=>

NULL-safe equal. Este operador realiza una comparación de igualdad como el operador =, pero retorna 1 en lugar de NULL si ambos operandos son NULL, y 0 en lugar de NULL si un operando es NULL.

```
SELECT 1 <=> 1, NULL <=> NULL, 1 <=> NULL;
SELECT 1 = 1, NULL = NULL, 1 = NULL;
```

1 <=> 1	NULL <=> NULL	1 <=> NULL	1 = 1	NULL = NULL	1 = NULL
1	1	0	1	*NULL*	*NULL*

Imagen 3. Aplicación No es Igual
Fuente: Propia.

- <>, != Diferente:

```
SELECT '.01' <> '0.01';
SELECT .01 <> '0.01';
SELECT 'zapp' <> 'zappp';
```

'.01' <> '0.01'	.01 <> '0.01'	'zapp' <> 'zappp'
1	0	1

Imagen 4. Aplicación Diferente
Fuente: Propia.

■ <= Menor que o igual:

```
SELECT 0.1 <= 2;
```

0.1 <= 2
1

Imagen 5. Aplicación Menor o igual
Fuente: Propia.

■ < Menor que:

```
SELECT 2 < 2;
```

2 < 2
0

Imagen 6. Aplicación Menor que
Fuente: Propia.

■ >= Mayor que o igual:

```
SELECT 2 >= 2;
```

2 >= 2
1

Imagen 7. Aplicación Mayor Igual
Fuente: Propia.

■ > Mayor que:

```
SELECT 2 > 2;
```

2 > 2
0

Imagen 8. Aplicación Mayor que
Fuente: Propia.

- IS valor booleano, IS NOT valor booleano

Comprueba si un valor contra un valor booleano, donde boolean_value puede ser TRUE, FALSE, o UNKNOWN.

```
SELECT 1 IS TRUE, 0 IS FALSE, NULL IS UNKNOWN;
SELECT 1 IS NOT UNKNOWN, 0 IS NOT UNKNOWN, NULL IS NOT UNKNOWN;
```

1 IS TRUE	0 IS FALSE	NULL IS UNKNOWN	1 IS NOT UNKNOWN	0 IS NOT UNKNOWN	NULL IS NOT UNKNOWN
1	1	1	1	1	0

Imagen 9. Aplicación Booleanos
Fuente: Propia.

- IS NULL, IS NOT NULL

Testea si un valor es o no NULL.

```
SELECT 1 IS NULL, 0 IS NULL, NULL IS NULL;
SELECT 1 IS NOT NULL, 0 IS NOT NULL, NULL IS NOT NULL;
```

1 IS NULL	0 IS NULL	NULL IS NULL	1 IS NOT NULL	0 IS NOT NULL	NULL IS NOT NULL
0	0	1	1	1	0

Imagen 10. Aplicación Is Not Null
Fuente: Propia.

- Operadores lógicos

Son los que comprueban la veracidad de una condición, regresando un dato de tipo BOOLEAN. Y el valor retornado es TRUE, FALSE o DESCONOCIDO.

Operador	Significado
=	Igual a
>	Mayor que
<	Menor que
>=	Mayor o igual que
<=	Menor o igual que
<>	No es igual a
!=	No es igual a
!<	No es menor que
!>	No es mayor que
NOT, !	NOT lógica
AND, &&	AND lógica
OR, ||	OR lógica
XOR	XOR lógica

Cuadro 1
Fuente: Propia.

Operador	Significado
1	TRUE
0	FALSE
NO NULL	NULL

Cuadro 2
Fuente: Propia.

Precedencias de los operadores

La precedencia de operadores se muestra en la siguiente lista, de menor a mayor precedencia. Los operadores que se muestran juntos en una línea tienen la misma precedencia.

Prioridad
:=
||, OR, XOR
&&, AND
NOT
BETWEEN, CASE, WHEN, THEN, ELSE
=, <=>, >=, >, <=, <, <>, !=, IS, LIKE, REGEXP, IN
|
&
<<, >>
-, +
*, /, DIV, %, MOD
^
!
BINARY, COLLATE

Cuadro 3
Fuente: Propia.

Paréntesis

El uso de paréntesis se utiliza aquí con el fin de realizar primero una operación específica contenida en este:

Símbolo	Significado
(...)	Paréntesis

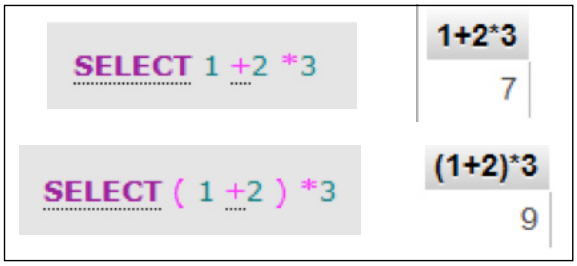


Imagen 11. Aplicación Parentesis
Fuente: Propia.

Funciones

Función	Descripción	Sintaxis
CREATE FUNCTION	Se utiliza para crear una función.	CREATE FUNCTION nombre_función (parametro1,parametro2,...) RETURNS tipoDato [Atributos de la rutina] <Bloque de instrucciones>
DROP FUNCTION	Se utiliza para borrar una función, especificamos el nombre de la función.	DROP FUNCTION nombre_funcion

Cuadro 4
Fuente: Propia.

Funciones de control de flujo

Función	Descripción	Sintaxis
IF	Se valida una condición la que puede tomar tres opciones si el primer argumento es verdadero, retorna el segundo argumento, sino retorna el tercero.	IF(expr1,expr2,expr3)
CASE	Similar a la anterior solo que esta establece varias condiciones a cumplir.	case when then ... else end
IFNULL	Si la condición se cumple retorna la Expresión, de lo contrario retorna expr2.	IFNULL(expr1,expr2)
NULLIF	Retorna NULL, si las valores en expr1 y expr2 son iguales delo contrario retorna expr1.	NULLIF(expr1,expr2)

Cuadro 5
Fuente: Propia.

Funciones de comparación de cadenas de caracteres

Se requiere comparar las cadenas de caracteres. Y estas retornan el valor NULL cuando su longitud es mayor que el valor de la variable de sistema. Entre las más importantes tenemos:

Función	Descripción	Sintaxis
ASCII(str)	Para valores numéricos de 0 a 255. Y Retorna el valor numérico del carácter más a la izquierda de la cadena de caracteres str.	SELECT ASCII( );
BIN(N)	Retorna una representación de cadena de caracteres del valor binario de N.	SELECT BIN( );
BIT_LENGTH(str)	Retorna la longitud de la cadena de caracteres str en bits.	SELECT BIT_LENGTH( );
CHAR(N,...)	Interpreta los argumentos como enteros y retorna la cadena de caracteres que consiste en los caracteres dados por los códigos de tales enteros.	SELECT CHAR( );
CHAR_LENGTH(str)	Retorna la longitud de la cadena de caracteres str, medida en caracteres.	SELECT CHAR_LENGTH()
CHARACTER_LENGTH(str)	Igual que el anterior.	SELECT CHARACTER_LENGTH(str)
COMPRESS(string_to_compress)	Comprime una cadena de caracteres.	SELECT LENGTH(COMPRESS(‘’));
CONCAT(str1,str2,...)	Retorna la cadena resultado de concatenar los argumentos.	SELECT CONCAT(4.9);
CONCAT_ WS(separator,str1,str2,...)	Evita valores NULL tras el argumento separador.	SELECT CONCAT_WS( );
CONV(N,from_base,to_base)	Convierte números entre diferentes bases numéricas.	SELECT CONV( );
ELT(N,str1,str2,str3,...)	Retorna la posición en la que se encuentra el valor deseado.	SELECT ELT(N, , , ,);
EXPORT_SET(bits,on,off[,separator[,number_of_bits]])	Retorna una cadena en que para cada bit del valor bits, puede obtener una cadena on y para cada bit reasignado obtiene una cadena off .	SELECT EXPORT_SET(,,,,,);
FIELD(str,str1,str2,str3,...)	Retorna el índice de str en la lista str1, str2, str3, ...	SELECT FIELD(, , , , ,);
FIND_IN_SET(str,strlist)	Retorna un valor en el rango de 1 a N si la cadena str está en la lista de cadenas strlist consistente de N subcadenas.	SELECT FIND_IN_SET(, , , ,);
HEX(N_or_S)	retorna una cadena representación del valor hexadecimal de N.	SELECT HEX( );
INSERT(str,pos,len,newstr)	Retorna la cadena str, con la subcadena comenzando en la posición pos y len caracteres reemplazados por la cadena newstr.	SELECT INSERT(, , ,);
INSTR(str,substr)	Retorna la posición de la primera ocurrencia de la subcadena substr en la cadena str.	SELECT INSTR(,);
LCASE(str)	Es sinónimo de LOWER().	
LEFT(str,len)	Retorna los len caracteres empezando por la izquierda de la cadena str.	SELECT LEFT(,);
LENGTH(str)	Retorna la longitud de la cadena str, medida en bytes. Un carácter multi-byte cuenta como múltiples bytes.	SELECT LENGTH( );
LOAD_FILE(file_name)	Debe estar localizado en el servidor, debe especificar la ruta completa al fichero, y debe tener el privilegio FILE.	LOAD_FILE(‘/tmp/picture’);
LOCATE(substr,str) , LOCATE(substr,str,pos)	La primera sintaxis retorna la posición de la primera ocurrencia de la subcadena substr en la cadena str. La segunda sintaxis retorna la posición de la primera.	SELECT LOCATE(,);
LOWER(str)	Retorna la cadena str con todos los caracteres cambiados a minúsculas.	SELECT LOWER( );
expr LIKE pat [ESCAPE ‘escape-char’]	Retorna 1 (TRUE) o 0 (FALSE). Si expr o pat es NULL, el resultado es NULL.	SELECT ‘Luis!’ LIKE ‘%D%v%’;

Cuadro 6

Fuente: Propia.

Funciones matemáticas

Validan una expresión y retornan NULL al encontrar error.

Función	Descripción	Sintaxis
ABS(*X*)	Retorna el valor absoluto de *X*.	SELECT ABS(X);
ACOS(*X*)	Retorna el arcocoseno de *X*.	SELECT ACOS(X);
ASIN(*X*)	Retorna el arcoseno de *X*.	SELECT ASIN(X);
ATAN(*X*)	Retorna la arcotangente de *X*.	SELECT ATAN(X);
ATAN(*Y*,*X*) , ATAN2(*Y*,*X*)	Retorna la arcotangente de las variables *X* y *Y*.	SELECT ATAN(-X,Y);
CEILING(*X*), CEIL(*X*)	Retorna el entero más pequeño no menor a *X*.	SELECT CEILING(X);
RAND(), RAND(*N*)	Retorna un valor aleatorio en coma flotante del rango de 0 a 1.0. Si se especifica un argumento entero *N*.	SELECT RAND();

Cuadro 7

Fuente: Propia.

Funciones de fecha y hora

Usualmente se requiere de funciones que permitan dar, la hora en diferentes formatos y convertir está en un dato deseado para realizar un cálculo. Veremos un listado de estas interesantes funciones.

Función	Descripción	Sintaxis
ADDDATE(date,INTERVAL expr type), ADDDATE(expr,days)	Cuando se invoca con la forma INTERVAL del segundo argumento, ADDDATE() es sinónimo de DATE_ADD().	SELECT DATE_ADD(, INTERVAL DAY);
ADDTIME(expr,expr2)	ADDTIME() añade expr2 a expr y retorna el resultado.	SELECT ADDTIME( );
CONVERT_TZ(dt,from_ tz,to_tz)	Convierte un valor datetime dt de la zona horaria dada por from_tz a la zona horaria dada por to_tz y retorna el valor resultante.	SELECT CONVERT_ TZ( );
CURRENT_DATE, CURRENT_ DATE()	Es sinónimos de CURDATE().	
CURTIME()	Retorna la hora actual como valor en formato 'HH:MM:SS' o HHMMSS.	SELECT CURTIME();
CURRENT_TIME, CURRENT_ TIME()	Es sinónimos de CURTIME().	
CURRENT_TIMESTAMP, CURRENT_TIMESTAMP()	Es sinónimos de NOW().	
DATE(expr)	Extrae la parte de fecha de la expresión de fecha o fecha y hora expr.	SELECT DATE( );
DATEDIFF(expr,expr2)	Retorna el número de días entre la fecha inicial expr y la fecha final expr2.	SELECT DATEDIFF( );
DATE_ADD(date,INTERVAL expr type), DATE_ SUB(date,INTERVAL expr type)	Realizan operaciones aritméticas de fechas.	SELECT DATE_ADD(, INTERVAL 1 DAY);

Cuadro 8

Fuente: Propia.

Funciones y operadores de cast y conver

Función	Descripción	Sintaxis
BINARY	Convierte la cadena a continuación a una cadena binaria.	SELECT BINARY 'x' = 'X';
CAST(expr AS type)	Toma un valor de un tipo y produce un valor de otro tipo.	CAST(expr AS type)
CONVERT(expr,type)	Toma un valor de un tipo y produce un valor de otro tipo.	CONVERT(expr,type)

Cuadro 9

Fuente: Propia.

Funciones bit

La aritmética que se aplica es la BIGINT. Con un total de 64 bit, utilizados para para operaciones de bit. Logrando un rango máximo de 64 bits.

Función	Sintaxis
| OR bit a bit	SELECT X | Y;
& AND bit a bit	SELECT X & Y;
^ XOR bit a bit	SELECT X ^ Y;
BIT_COUNT(N)	SELECT BIT_COUNT(N);

Cuadro 10
Fuente: Propia.

Funciones de encriptación

A menudo se utilizan para encriptar y desencriptar valores.

Función	Descripción	Sintaxis
AES_ENCRYPT	Si detecta datos inválidos retorna NULL. Pero si estos son válidos el valor a retornar no NULL.	AES_ENCRYPT(str,key_str)
DECODE	Desencripta la cadena encriptada crypt_str usando pass_str como contraseña.	DECODE(crypt_str,pass_str)
ENCODE	Encripta str usando pass_str como contraseña.	ENCODE(str,pass_str)
DES_DECRYPT	Desencripta una cadena encriptada.	DES_DECRYPT(crypt_str[,key_str])
DES_ENCRYPT	Encriptas la cadena con la clave dada usando el algoritmo triple-DES.	DES_ENCRYPT(str[,(key_num|key_str)])
ENCRYPT	Encripta str usando la llamada de sistema Unix crypt().	ENCRYPT(str[,salt])
MD5	Calcula una checksum MD5 de 128-bit para la cadena.	MD5(str)
OLD_PASSWORD	Retorna el valor de la implementación pre-4.1 de PASSWORD().	OLD_PASSWORD(str)
PASSWORD	Calcula y retorna una cadena de contraseña de la contraseña en texto plano str, o NULL si el argumento era NULL.	PASSWORD(str)
SHA1(str)	Calcula una checksum SHA1 de 160-bit para la cadena.	SELECT SHA1('xxx');
SHA(str)	Calcula una checksum SHA de 160-bit para la cadena.	SELECT SHA1('xxx');

Cuadro 11
Fuente: Propia.

Funciones de información

Función	Descripción	Sintaxis
BENCHMARK	Ejecuta la expresión expr repetidamente.	BENCHMARK(count,expr)
CHARSET(str)	Retorna el conjunto de caracteres el argumento cadena.	SELECT CHARSET('xyz');
COLLATION(str)	Retorna la colación para el conjunto de caracteres de la cadena dada.	SELECT COLLATION('xyz');
CONNECTION_ID()	Retorna el ID de la conexión.	SELECT CONNECTION_ID();
CURRENT_USER()	Retorna la combinación de nombre de usuario y de equipo que tiene la sesión actual.	SELECT CURRENT_USER();
DATABASE()	Retorna el nombre de base de datos por defecto.	SELECT DATABASE();
FOUND_ROWS()	Restringir el número de registros que el servidor retorna al cliente.	SELECT FOUND_ROWS();
LAST_INSERT_ID(expr)	Retorna el último valor generado automáticamente que se insertó en una columna AUTO_INCREMENT.	LAST_INSERT_ID()
ROW_COUNT();	Retorna el número de registros actualizados, insertados o borrados por el comando precedente.	SELECT ROW_COUNT();
USER()	Retorna el nombre de usuario y de equipo.	SELECT USER();
SESSION_USER()	Es sinónimo de USER().	SELECT SESSION_USER();
SYSTEM_USER()	Es sinónimo de USER().	SELECT SYSTEM_USER() ;
VERSION()	Retorna una cadena que indica la versión del servidor.	SELECT VERSION();

Cuadro 12
Fuente: Propia.

Funciones varias

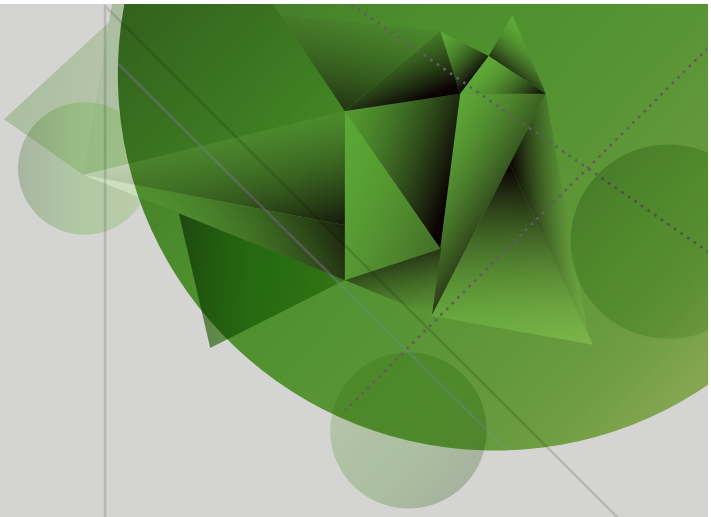
Función	Descripción	Sintaxis
DEFAULT(col_name)	Retorna el valor por defecto para una columna de tabla.	UPDATE c SET j = DEFAULT(j)+1 WHERE id < 25;
FORMAT(H,J)	Formatea el número H a un formato como '#,###,###.##', redondeado a J decimales, y retorna el resultado como una cadena.	SELECT FORMAT(45632.456789, 7);
GET_LOCK(str,timeout)	Intenta obtener un bloqueo con un nombre dado por la cadena str, con un tiempo máximo de timeout segundos.	SELECT GET_LOCK('lock1',50);
INET_ATON(expr)	Dada la representación de cuatros números separados por puntos de una dirección de red como cadena de caracteres, retorna un entero que representa el valor numérico de la dirección.	SELECT INET_ATON('192.168.0.23');
INET_NTOA(expr)	Dada una dirección de red numérica (4 o 8 bytes), retorna la representación de puntos de la dirección como cadena.	SELECT INET_NTOA(192168023);
IS_FREE_LOCK(str)	Chequea si el nombre de bloqueo str está libre para uso. Y retorna 1, 0 o null.	IS_FREE_LOCK( )
IS_USED_LOCK(str)	Chequea si el nombre de bloqueo str está en uso (esto es, bloqueado).	IS_USED_LOCK( )
MASTER_POS_WAIT(log_name,log_pos[,timeout])	Esta función es útil para control de sincronización de maestro/ esclavo.	MASTER_POS_WAIT(log_name,log_pos[,timeout])
RELEASE_LOCK(str)	Libera el bloqueo nombrado por la cadena str obtenida con GET_LOCK().	SELECT UUID();

Cuadro 13
Fuente: Propia.

2

Unidad 2

Transacciones en MySQL



Bases de datos II

Autor: Julio Alberto Castillo

AREANDINA
Fundación Universitaria del Área Andina

MIEMBRO DE LA RED
ILUMNO

Introducción

En esta semana la cartilla está dirigida a conocer los conceptos respecto a los procesos que se desarrollan cuando se realizan transacciones en MySQL, esto con el fin de que los estudiantes tengan pleno conocimiento de su funcionamiento y cuáles son las maneras más efectivas de utilizar este manejo y que además entiendan todas las posibles causas y efectos de su uso.

Todo este proceso será realizado dando ejemplos que puedan de aluna manera ilustrar al estudiante, además de secuencias de instrucciones que pueden ejecutarse desde MySQl y que se pueda corroborar de manera efectiva su acción dándole confiabilidad al proceso y además estableciendo y arraigando conceptos que serán de gran utilidad cuando estén siendo usados en la vida cotidiana.

Cada una de las instrucciones será explicada de manera detallada teniendo en cuenta cada componente y variación para que podamos observar cuales son las diferentes aplicaciones y cuál es el efecto que tiene en la base de datos y en la información esto con el fin de generar modelos efectivos e incentivar al estudiante a realizar sus propia prácticas y que de manera autónoma y basado en elementos cotidianos que se le presenten en su medio académico o laboral pueda resolver o plantear por lo menos una solución efectiva.

U2

Metodología

Con el fin de que el estudiante realice la mayor aprensión del conocimiento se realizaran las siguientes recomendaciones metodológicas:

Realizar las lecturas complementarias las cuales le permitirán ampliar conceptos y comprender la temática tratada en la Unidad

Utilizar fuentes bibliográficas e información de internet, recolectada para una mayor comprensión de la información sobre los temas propuestos.

Clasificar la información recolectada y realizar modelos aplicativos en los cuales el estudiante pueda corroborar y experimentar el funcionamiento y las diferentes maneras que hay para trabajar las múltiples funcionalidades que tiene el lenguaje SQL.

U2

Desarrollo temático

Transacciones en MySQL

Transacciones y niveles de aislamiento en MySQL

Lo primero que debemos comenzar a entender son los conceptos relacionados con la temática de esta semana por eso comenzaremos con algunas definiciones.

¿Qué es una transacción?

Son un conjunto de órdenes que se ejecutan en un sistema gestor de base de datos como son (MySQL, SQL Server, Oracle, PostgreSQL, Sybase, DB2) formando una unidad integral de trabajo (a esto se le denomina atomicidad)

MySQL es un sistema, a **transaccional**, porque tiene la capacidad de mantener la integridad de los datos, haciendo que las transacciones no terminen en un estado intermedio. Es por esto que se le atribuye el acrónimo ACID.

Si por alguna causa el sistema debe cancelar una transacción, empieza a deshacer las sentencias ejecutadas hasta dejar la base de datos en su estado inicial (**llamado punto de integridad**), esto le permite al sistema mantener la información estable y segura además de actualizada.

Conceptos asociados a la sigla ACID

- Atomicidad: es el proceso que realiza una transacción que no es tangible, o sea, que deben ejecutarse todas las instrucciones de una transacción como una unidad de trabajo la cual no puede ser dividida, en caso de que alguna de las instrucciones falle no se debe ejecutar ninguna indiferentemente de la cantidad.
- Coherencia: este término es muy importante ya que indica que únicamente datos válidos pueden ser almacenados en la base. Si se ejecuta una transacción que compromete la coherencia interna de la base de datos, toda la transacción debe ser cancelada.
- Aislamiento: las transacciones que se realicen de manera simultánea deben ejecutarse separadas unas de otras hasta que cada una de ellas finalicen.
- Durabilidad: mantener la integridad de una transacción una vez haya sido confirmada significa que ya no podrá ser eliminada por acción de la atomicidad.

Posibles causas para que una transacción sea cancelada

- Alta concurrencia a una base de datos o unas tablas específicas (Ingreso simultáneo de muchos datos por los mismos o diferentes usuarios).
- Algunas ejecuciones de instrucciones paralelas pueden mezclarse de manera furtiva y pueden dejar a la base de datos en un estado inconsistente.
- Falla del sistemas operativo.
- Falla de energía eléctrica.
- Falla en el software de base de datos.

Ejecuciones en serie

Para entender este concepto es mejor colocar un ejemplo que nos permita entender su naturaleza.

Ejemplo: en un sistema de subastas en línea, dos usuarios están ofertando por el mismo producto, como es natural alguno de los dos debe ganar, cuando los usuarios están en el sistema, se pueden ver los siguientes procesos:

- El sistema buscará productos que puedan ser ofrecidos y pueda llevar el usuario y además que esté disponible.
- El usuario elige el producto.
- El sistema asigna el producto al usuario que mejor oferta haya realizado.

Es posible que ambos usuarios hayan realizado la misma oferta por el mismo producto y el sistema se los haya permitido dejando a la base de datos en un **estado indeseable**.

Entonces ahora explicaremos en que consiste el proceso de transacciones en serie:

1. El estado de la base de datos debe quedar como si una operación fue realizada primero y otra después o sea de manera secuencial a esto se le denomina ejecuciones en serie).
2. Si una ejecución es en serie, nunca se le podrá asignará a los dos usuarios el mismo producto cuyo valor ha sido ofertado de manera superior.
3. Se debe tener en cuenta que no se desea que un proceso se realice de manera secuencial, pero si se necesita que el resultado sea en serie.

Atomicidad

Este concepto también lo explicaremos con un ejemplo.

Ejemplo: En un proceso de transferencia de dinero entre dos cuentas Cuenta1 y Cuenta2 se realizan las siguientes acciones:

- Verificar que Cuenta1 tenga suficiente dinero para trasferir.

- Aumenta el saldo de Cuenta2 en la cantidad trasferida.
- Disminuir el saldo de Cuenta1 en la cantidad trasferida.

Supongamos que el sistema presenta una falla antes de ejecutar la tercera tarea, lo que genera que la base de datos se encuentre en un estado indeseable.

Entonces la atomicidad consiste en que todas las operaciones en este caso tres se ejecuten o que ninguna de ellas lo haga.

¿El uso de transacciones resuelve los problemas de atomicidad y ejecuciones en serie?

En efecto, porque una transacción está compuesta por un grupo de instrucciones SQL que se ejecutan todas o ninguna de ellas y además se les exige ejecuciones en serie.

Partes de una transacción

1. Toda transacción comienza con la sentencia begin. Esta sentencia indica a la base de datos que debe estar preparada porque se inicia la entrada de un conjunto de instrucciones SQL que la modificaran.
2. Ejecución y validación del conjunto de sentencias SQL que modificarán la base de datos.
3. Aquí existen dos posibilidades:
   - Si todo el proceso se realizó de manera satisfactoria, entonces se debe ejecutar la instrucción commit la cual tiene como objetivo que la transacción se grabe de forma permanente.
   - Si hubo algún problema en el proceso, entonces se debe ejecutar la instrucción rollback la cual cancela la transacción y hace que sea no exitosa, por lo tanto cualquier cambio que la transacción haya podido hacer a la BD se deshace para mantener la integridad.

Del ejemplo de transferencia de fondos entre cuentas entonces debería quedar algo así:

1. **begin**
1. La cuenta1 no tiene suficientes fondos --> **rollback**
2. Se aumenta el saldo de Cuenta2 al monto trasferido.
3. Se disminuye el saldo de Cuenta1 en el monto traferido.
1. **commit**

Niveles de aislamiento en transacciones

SQL permite definir diferentes niveles de aislamiento para el tratamiento de las transacciones.

- Serializable.

- Read Commited.
- Repeatable Read.
- Read Uncommited.

Para entenderlo mejor, usaremos un ejemplo:

Ejemplo:

- La tienda de Julio vende dos tipos de bebidas energéticas Red Bull y Satan a $3.500 y $4.000 respectivamente.
- Juan hace una consulta sobre la bebida más barata y sobre la bebida más cara.
- Julio al mismo tiempo que Juan hace la consulta modifica la base de datos, eliminando ambas marcas de cerveza pero ingresando una nueva marca Vive100 a $5.000.
- Juan ejecuta las siguientes consultas.

```
1 Select max(precio) from venta where tienda="Julio";
2
3 Select min(precio) from venta where tienda="Julio";
```

Imagen 1. Select General
Fuente: Propia.

- A estas consultas les llamaremos (max) y (min) respectivamente.
- Por su parte Julio ejecuta.

```
1 delete from venta where tienda = "Julio";
2
3 insert into venta values ("Julio","Vive 100",5000);
```

Imagen 2. Select General
Fuente: Propia.

- A estas consultas les llamaremos (borrado) e (ingreso) respectivamente.
- Supongamos que ambas consultas se ejecutan simultáneamente en la base de datos.
- Lo único que podemos asegurar es que (mayor) se ejecutó antes que (menor) y que (borrado) se ejecutó antes que (ingreso).
- Muestro una imagen de una posible ejecución.

Juan	Mayor			Menor
Julio		Borrado	Ingreso	

Cuadro 1
Fuente: Propia.

- Juan lee que el precio máximo es de la bebida Satan a $4.000 y lee como mínimo el precio de Vive100 a $5.000.

Nivel serializable

Si Juan ejecuta sus instrucciones con una base de datos MySQL con un nivel de aislamiento serializable, entonces la base de datos responderá con datos antes o después de la ejecución de las instrucciones de Julio pero nunca en el medio. Por lo tanto con esto nos aseguramos que un grupo de instrucciones se ejecuten antes y otro después.

Juan	Mayor	Menor		
Julio			Borrado	Ingreso

Cuadro 2
Fuente: Propia.

Se le consideran como el nivel máximo de aislamiento y también genera el máximo nivel de bloqueos.

Imagen 3. Resultado comando
Fuente: Propia.

Nivel Read Commited (lecturas confirmadas)

Este nivel de aislamiento evita la lectura sucia de datos. Este nivel hace que SGBD lea y devuelva información que ha sido confirmada.

Por ejemplo, Julio ejecuta (Borrado) e (ingreso) pero luego lo piensa, se arrepiente y hace Rollback para deshacer los cambios.

Si Juan ejecuta su consulta después del (ingreso) y antes del Rollback.

Juan			Mayor	Menor
Julio	Borrado	Ingreso		

Cuadro 3
Fuente: Propia.

Juan lee el precio $5.000 como máximo y mínimo, sin embargo $5.000 es un dato que nunca existirá (lectura sucia). Este nivel evita este tipo de lecturas ya que nunca fue confirmada.

Los problemas de este nivel son:

- Lecturas no repetibles: dos sentencias SELECT iguales y consecutivas podrían devolver datos diferentes.
- Datos fantasmas: dos sentencias SELECT iguales y consecutivas podrían aparecer y desaparecer filas.

Otra posibilidad de lectura sucia es que si Julio hace commit, Juan lea como máximo $4.500 y como mínimo $5.000 si las consultas se realizan de la siguiente forma.

Juan	Mayor			Menor
Julio		Borrado	Ingreso	

Cuadro 4
Fuente: Propia.

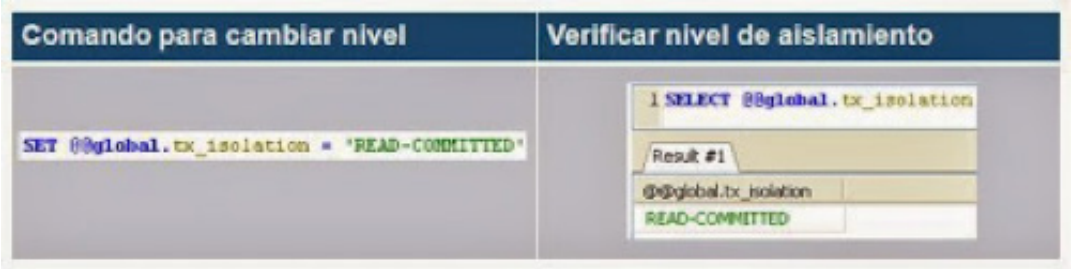


Imagen 4. Resultado comando
Fuente: Propia.

Nivel Repeatable Read (lectura repetible)

Este nivel de aislamiento garantiza que dos consultas consecutivas diferentes dentro de una transacción devolverán información consistente.

Supongamos que Juan ejecuta sus consultas sobre una base de datos MySQL con nivel de aislamiento Repeatable Read y el orden de sus consultas es:

Juan	Mayor			Menor
Julio		Borrado	Ingreso	

Cuadro 5
Fuente: Propia.

Durante las lecturas (mayor), Juan leyó $3.500 y $4.000, el SGBD debe asegurar que durante (menor) se vean adicionalmente a $5.000, los valores de $3.500 y $4.000 ya que estos fueron vistos en la lectura anterior, por lo que Juan verá datos consistentes: máximo precio es $4.000 y el mínimo es $3.500 aunque esto no refleje el estado actual de la base de datos.

El problema de este nivel son los datos fantasma: dos sentencias SELECT iguales y consecutivas podrían aparecer datos diferentes.

Por ejemplo Juan intenta leer dos veces el precio máximo (mayor).

Juan	Mayor			Mayor
Julio		Borrado	Ingreso	

Cuadro 6
Fuente: Propia.

Si la consulta es ejecutada cuando la base de datos se encuentra con el nivel de aislamiento **repeatable read** se asegura que todo lo que lee en el primer (mayor) lo lee también en el segundo (mayor), sin embargo en un caso obtiene que el máximo es $4.000 y luego $5.000.

Imagen 5. Resultado comando
Fuente: Propia.

Nivel Read Uncommited (lectura no confirmada)

Este nivel es el menos aconsejable para muchos casos, pero esto no quiere decir que para otros métodos no sirva.

Los problemas de este nivel es que permite: lecturas sucias, lecturas no repetibles y lecturas fantasmas.

Imagen 6. Resultado comando
Fuente: Propia.

Sintaxis de START TRANSACTION, COMMIT y ROLLBACK

Cuando usamos MySQL el modo autocommit está activado. Esto significa que en cuanto ejecute cualquier comando que actualice o modifique (update) una tabla, MySQL almacena la actualización inmediatamente.

Si usa tablas transaccionales con formatos (InnoDB o BDB), se puede desactivar el modo autocommit usando el siguiente comando:

```
SET AUTOCOMMIT=0
```

Cuando se deshabilita el modo autocommit colocando la variable AUTOCOMMIT en cero, necesariamente debe usar la instrucción COMMIT para poder almacenar los cambios en la base o la instrucción ROLLBACK si requiere ignorar los cambios hechos desde el inicio de la transacción.

Si por alguna circunstancia se requiere deshabilitar el modo autocommit para una serie exclusiva de comandos, puede usar el comando START TRANSACTION el cual tiene la siguiente sintaxis:

```
1 START TRANSACTION;
2 SELECT @A:=SUM(salary) FROM table1 WHERE type=1;
3 UPDATE table2 SET summary=@A WHERE type=1;
4 COMMIT;
```

Imagen 7. Resultado comando
Fuente: Propia.

Con START TRANSACTION, el autocommit es deshabilitado hasta llegar al final de la transacción y asegurarla con COMMIT o deshacerla con ROLLBACK. En ese instante el modo autocommit vuelve a su estado original.

BEGIN y BEGIN WORK se soportan como alias para START TRANSACTION sirven para iniciar una transacción. START TRANSACTION tiene sintaxis SQL estándar y es la forma que se recomienda para iniciar una transacción. Por el contrario el comando BEGIN difiere del uso de la palabra clave BEGIN que comienza un comando BEGIN... END. Este último no puede comenzar una transacción. La manera para comenzar una transacción seria así:

START TRANSACTION WITH CONSISTENT SNAPSHOT

La cláusula WITH CONSISTENT SNAPSHOT da lugar a una lectura consistente únicamente para motores de almacenamiento que tengan la capacidad de soportarlo. Actualmente, esto se aplica sólo al formato InnoDB. El efecto obtenido es el mismo que realizar un START TRANSACTION seguido por un SELECT desde cualquier tabla que tenga formato InnoDB.

Comenzar una transacción provoca que se realice un UNLOCK TABLES implícito.

Tenga en cuenta que si no usa tablas transaccionales, cualquier cambio que se realice se almacena de manera inmediata, a pesar del estado en que puedas tener el modo autocommit.

Si realiza la ejecución de un comando ROLLBACK tras realizar la actualización a una tabla que no tenga formato transaccional, de inmediato se genera una advertencia E**R_WARNING_NOT_COMPLETE_ROLLBACK**. La cual nos indica que los cambios realizados en tablas transaccionales se deshacen, pero por el contrario los cambios realizados en tablas no transaccionales van a permanecer.

Cada transacción que se realiza se almacena en el log binario en un sector, hasta cuando se ejecute el comando COMMIT. Las transacciones que se deshacen no se validan. (Excepción: Las modificaciones a tablas no transaccionales no pueden deshacerse. Si una transacción que se deshace incluye modificaciones a tablas no transaccionales, la transacción entera se valida con un comando ROLLBACK al final para asegurar que las modificaciones a estas tablas se realizaron.) Otra manera es cambiar el nivel de aislamiento para transacciones con SET TRANSACTION ISOLATION LEVEL.

Deshacer transacciones es normalmente una operación lenta que puede ocurrir sin que el usuario lo haya pedido explícitamente (por ejemplo, cuando por cualquier razón ocurre un error). Debido a ello, SHOW PROCESSLIST en MySQL muestra Rolling back durante rollbacks implícitos y explícitos (comando SQL ROLLBACK).

Características de las sentencias que no se pueden deshacer

Algunos comandos no tienen posibilidad de deshacerse. En general, esto incluye comandos del lenguaje de definición de datos (DDL), tales como los que crean y borran bases de datos, los que crean, borran o alteran tablas o rutinas de instrucciones almacenadas.

Debe verificar que sus transacciones no incluyan tales comandos. Si realiza un comando en una transacción que no puede deshacerse, y luego un comando enviado de forma posterior falla, el efecto general de la transacción no puede deshacerse mediante un comando ROLL-BACK.

Sentencias que causan una ejecución (COMMIT) implícita (Automática)

Cada uno de los siguientes comandos y cualquier otro que tenga características similares, terminan una transacción implícitamente, como si hubiera ejecutado un COMMIT antes de ejecutar el comando:

LOAD MASTER DATA	LOCK TABLES	RENAME TABLE
CREATE TABLE		CREATE DATABASE
DROP DATABASE	DROP INDEX	DROP TABLE
SET AUTOCOMMIT=1	START TRANSACTION	TRUNCATE TABLE
ALTER TABLE	BEGIN	CREATE INDEX

Cuadro 7
Fuente: Propia.

UNLOCK TABLES también realiza un COMMIT de una transacción si existe o detecta cualquier tabla bloqueada.

Ojo con esto las transacciones no pueden anidarse. Esto es una consecuencia del COMMIT implícito realizado por cualquier transacción actual cuando se realiza un comando START TRANSACTION o uno de sus sinónimos.

Sintaxis de las instrucciones SAVEPOINT y ROLLBACK TO SAVEPOINT

```
1 SAVEPOINT identificado
2 ROLLBACK TO SAVEPOINT identificador
```

Imagen 8. Resultado comando
Fuente: Propia.

En la base de datos de MySQL, InnoDB soporta los comandos SAVEPOINT y ROLLBACK TO SAVEPOINT.

El comando SAVEPOINT crea un punto de validación o control dentro de una transacción con un nombre cualquiera en este caso identificador. Si la transacción actual tiene otro punto con el mismo nombre, el antiguo se borra y se crea el nuevo.

El comando ROLLBACK TO SAVEPOINT deshace una transacción hasta el punto que se ha definido. Las modificaciones que la transacción actual hace al registro tras el punto de control se deshacen en el rollback, pero InnoDB no libera los bloqueos de registro que se almacenaron en memoria tras el punto de control. (Tenga en cuenta que para un nuevo registro insertado, la información de bloqueo se realiza a partir del ID de transacción almacenado en el registro; el bloqueo no se almacena de manera separada en memoria. En este caso, el bloqueo de registro se libera al deshacerse todo.) Los puntos que se crean después del punto definido se borran.

Si un comando retorna el siguiente error, significa que no existe ningún punto con el nombre especificado:

```
ERROR 1181: Got error 153 during ROLLBACK
```

Todos los puntos de la transacción actual se borran si ejecuta un COMMIT, o un ROLLBACK que no nombre ningún punto de control.

Sintaxis de LOCK TABLES y UNLOCK TABLES (Bloqueo y Desbloqueo)

LOCK TABLES

nombre_tabla [AS alias] {READ [LOCAL] | [LOW_PRIORITY] WRITE}

[,nombre_tabla [AS alias] {READ [LOCAL] | [LOW_PRIORITY] WRITE}] ...

UNLOCK TABLES

Ejemplo

```
START TRANSACTION;
LOCK TABLE t_tabla WRITE;
INSERT INTO t_tabla VALUES (1,'2007-09-15');
INSERT INTO t_tabla VALUES (4,'2007-09-15');
UNLOCK TABLES;
COMMIT;
```

LOCK TABLES realiza el bloqueo de tablas para el flujo actual de instrucciones. Si alguna de las tablas la bloquea otro flujo de instrucciones, este las bloquea hasta que pueden realizarse todos los bloqueos. UNLOCK TABLES libera cualquier bloqueo realizado por el flujo de instrucciones actual. Todas las tablas bloqueadas por el flujo de instrucciones actual se liberan automáticamente cuando el flujo realiza otro LOCK TABLES, o cuando la conexión con el servidor se cierra.

Un bloqueo de tabla protege únicamente contra lecturas inapropiadas o escrituras de otros usuarios. El usuario que tenga el bloqueo, incluso un bloqueo de lectura, puede realizar otras operaciones a nivel de tabla tales como un DROP TABLE lo que acarrearía serias consecuencias.

Tenga en cuenta las siguientes advertencias a pesar del uso de LOCK TABLES con tablas transaccionales:

- LOCK TABLES no es una operación transaccional y hace un COMMIT implícito de cualquier transacción activa antes de tratar de bloquear las tablas. También, comenzar una transacción (por ejemplo, con la instrucción START TRANSACTION) realiza un UNLOCK TABLES implícito.
- La forma correcta de usar LOCK TABLES con tablas transaccionales, como InnoDB, es poner AUTOCOMMIT = 0 y no realizar llamados de UNLOCK TABLES hasta que se realice un COMMIT de la transacción explícitamente. Cuando llama a LOCK TABLES, InnoDB internamente realiza su propio bloqueo de tabla y MySQL realiza su propio bloqueo de tabla. InnoDB libera su bloqueo de tabla en el siguiente COMMIT, pero para que MySQL libere su bloqueo de tabla, debe llamar a UNLOCK TABLES. No debe tener AUTOCOMMIT = 1, porque entonces InnoDB libera su bloqueo de tabla inmediatamente tras la llamada de LOCK TABLES, y los deadlocks pueden ocurrir fácilmente. (Tenga en cuenta que en MySQL, no adquirimos el bloqueo de tabla InnoDB en absoluto si AUTOCOMMIT=1, para ayudar a aplicaciones anteriores a evitar deadlocks).
- ROLLBACK no libera bloqueos de tablas no transaccionales de MySQL.

Para usar LOCK TABLES en MySQL, debe tener el permiso LOCK TABLES y el permiso SELECT para las tablas involucradas.

La razón principal para usar LOCK TABLES es para emular transacciones o para obtener más velocidad al actualizar tablas. Esto se explica con más detalle posteriormente.

Si un flujo obtiene un bloqueo READ en una tabla, ese flujo (y todos los otros) sólo pueden leer de la tabla. Si un flujo obtiene un bloqueo WRITE en una tabla, sólo el flujo con el bloqueo puede escribir a la tabla. El resto de flujos se bloquean hasta que se libera el bloqueo.

La diferencia entre READ LOCAL y READ es que READ LOCAL permite comandos INSERT no conflictivos (inserciones concurrentes) se ejecuten mientras se mantiene el bloqueo. Sin embargo, esto no puede usarse si va a manipular los ficheros de base de datos fuera de MyS-

QL mientras mantiene el bloqueo. Para tablas InnoDB, READ LOCAL esencialmente no hace nada: No bloquea la tabla. Para tablas InnoDB, el uso de READ LOCAL está obsoleto ya que una SELECT consistente hace lo mismo, y no se necesitan bloqueos.

Cuando usa LOCK TABLES, debe bloquear todas las tablas que va a usar en sus consultas. Mientras los bloqueos obtenidos con un comando LOCK TABLES están activas, no puede acceder a ninguna tabla que no estuviera bloqueada por el comando. Además, no puede usar una tabla bloqueada varias veces en una consulta para poder realizar este proceso debe usar alias en cada una de ellas. Tenga en cuenta que en este caso, debe tener un bloqueo separado para cada alias que haya utilizado.

```
1  LOCK TABLE tabla WRITE, tabla AS t1 WRITE;
2  INSERT INTO tabla SELECT * FROM tabla;
3  Table 'tabla' was not locked with LOCK TABLES
4  INSERT INTO tabla SELECT * FROM tabla AS t1;
```

Imagen 9. Resultado comando bloqueo insert
Fuente: Propia.

Si sus consultas se refieren a una tabla que use un alias, debe realizar el bloqueo de la tabla que usa el mismo alias. Esto No funciona si se realiza el bloqueo de la tabla sin especificar el alias:

```
1 LOCK TABLE tabla READ;
2 SELECT * FROM tabla AS tablaalias;
3 ERROR 1100: Table 'tablaalias' was not locked with LOCK TABLES
4
```

Imagen 10. Resultado comando bloqueo
Fuente: Propia.

Si bloquea una tabla usando un alias, debe referirse a ella en sus consultas usando el alias que haya definido:

```
1 LOCK TABLE tabla AS tablaalias READ;
2 SELECT * FROM tabla;
3 ERROR 1100: Table 'tabla' was not locked with LOCK TABLES
4 SELECT * FROM tabla AS tablaalias;
```

Imagen 11. Resultado Bloqueo tablas
Fuente: Propia.

WRITE bloquea normalmente teniendo una prioridad más alta que READ al bloquear para así asegurar que las actualizaciones que se realicen, sean procesadas en cuanto se pueda. Esto significa que si un flujo obtiene un bloqueo READ y luego otro flujo pide un bloqueo WRITE, las peticiones realizadas por el bloqueo READ posteriores esperan hasta que el flujo WRITE quita el bloqueo. Puede usar bloqueos LOW_PRIORITY WRITE para permitir a otros flujos instruccionales que obtengan bloqueos READ mientras el flujo está en espera para el bloqueo WRITE. Debe usar bloqueos LOW_PRIORITY WRITE sólo si está seguro que habrá un momento sin flujos con bloqueos READ.

El funcionamiento de LOCK TABLES es el siguiente:

1. Ordena todas las tablas a ser bloqueadas en un orden definido internamente. Desde el punto de vista del usuario, este orden es indefinido.
2. Si una tabla se bloquea con bloqueo de escritura y lectura, coloca el bloqueo de lectura antes del de escritura.
3. Bloquea una tabla cada vez hasta que el flujo de información obtiene todos los bloqueos.

Esta política asegura un bloqueo de tablas libre de deadlocks. Sin embargo hay otros puntos que debe tener en cuenta respecto a esta directriz:

Si está usando un bloqueo LOW_PRIORITY WRITE para una tabla, sólo significa que MySQL espera para este bloqueo, hasta que no hay flujos que quieren un bloqueo READ. Cuando el flujo ha obtenido el bloqueo WRITE y está esperando para obtener un bloqueo para la siguiente tabla en la lista, todos los otros flujos esperan hasta que el bloqueo WRITE se libera. Si esto es un problema con su aplicación, debe considerar convertir algunas de sus tablas a transaccionales.

Puede usar KILL para terminar un flujo que está esperando para un bloqueo de tabla.

Tenga en cuenta que no debe bloquear ninguna tabla que esté usando con INSERT DELAYED ya que en tal caso el INSERT lo realiza un flujo de instrucciones separado.

Normalmente, no tiene que bloquear tablas, ya que todos los comandos UPDATE son atómicos, ningún otro flujo puede interferir con ningún otro que está ejecutando comandos SQL. Hay algunos casos en que no se deben bloquear tablas de ninguna manera:

- Si va a ejecutar varias operaciones en un conjunto de tablas MyISAM, es mucho más rápido bloquear las tablas que va a usar. Bloquear tablas MyISAM acelera la inserción, las actualizaciones, y los borrados. Por el contrario, ningún flujo puede actualizar una tabla con un bloqueo READ (incluyendo el que tiene el bloqueo) y ningún flujo puede acceder a una tabla con un bloqueo WRITE distinto al que tiene el bloqueo.

  La razón que algunas operaciones MyISAM sean más rápidas bajo LOCK TABLES es que MySQL no vuelca la caché de claves para la tabla bloqueada hasta que se llama a UNLOCK TABLES. Normalmente, la caché de claves se vuelca tras la ejecución de cada comando SQL.

- Si usa un motor de almacenamiento en MySQL que no soporta transacciones, debe usar LOCK TABLES si quiere asegurarse que ningún otro flujo se ejecute entre un SELECT y un UPDATE. El ejemplo mostrado necesita LOCK TABLES para ejecutarse sin problemas:

```
1 LOCK TABLES transaccion READ, personas WRITE;
2 SELECT SUM(value) FROM transaccion WHERE persona_id=id;
3 UPDATE personas
4 SET total_value=sum_from_previous_statement
5     WHERE persona_id=id;
6 UNLOCK TABLES;
```

Imagen 12. Resultado comando
Fuente: Propia.

Sin LOCK TABLES, es posible que otro flujo pueda insertar un nuevo registro en la tabla transacción entre la ejecución del comando SELECT y UPDATE.

Puede evitar usar LOCK TABLES en varios casos usando actualizaciones relativas (UPDATE persona SET value=value+new_value) o la función LAST_INSERT_ID() , Puede evitar bloquear tablas en algunos casos usando las funciones de bloqueo de nivel de usuario GET_LOCK() y RELEASE_LOCK(). Estos bloqueos se guardan en una tabla hash en el servidor e implementa pthread_mutex_lock() y pthread_mutex_unlock() para alta velocidad.

Puede bloquear todas las tablas en todas las bases de datos con bloqueos de lectura con el comando FLUSH TABLES WITH READ LOCK. Esta es una forma muy conveniente para obtener copias de seguridad si tiene un sistema de archivos que puede obtener el estado en un punto temporal.

Nota: Si usa ALTER TABLE en una tabla bloqueada, esta puede desbloquearse.

Sintaxis de SET TRANSACTION

SET [GLOBAL | SESSION] TRANSACTION ISOLATION LEVEL

{READ UNCOMMITTED | READ COMMITTED | REPEATABLE READ | SERIALIZABLE }

```
START TRANSACTION;
INSERT INTO Test VALUES (5), (6);
INSERT INTO Test VALUES (7), (8);
ROLLBACK;
SELECT * FROM Test;
```

Este comando prepara el nivel de aislamiento de una transacción para la siguiente transacción de forma global, o para la sesión actual.

El comportamiento por defecto de SET TRANSACTION es colocar el nivel de aislamiento para la siguiente transacción (la cual no ha iniciado todavía). Si usa la palabra clave GLOBAL el comando pone el nivel de aislamiento de transacción por defecto global para todas las transacciones creadas desde ese momento. Las conexiones existentes no se ven afectadas. Necesita el permiso de SUPER para hacerlo. Usar la palabra clave SESSION determina el nivel de transacción para todas las transacciones futuras pero realizadas en la conexión actual.

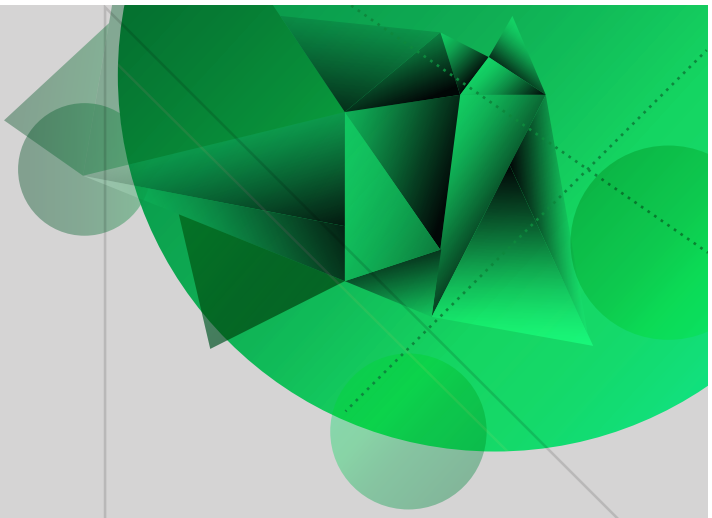
Para descripciones del nivel de aislamiento de cada transacción InnoDB. InnoDB soporta cada uno de estos niveles en MySQL. El nivel por defecto es REPEATABLE READ[1].

1 Adaptado de la documentación de la página oficial de MySQL. http://dev.mysql.com/doc/refman/5.6/en/

3

Unidad 3

Control de concurrencia



Bases de datos II

Autor: Julio Alberto Castillo

AREANDINA
Fundación Universitaria del Área Andina

MIEMBRO DE LA RED
ILUMNO

Introducción

Los procesos de concurrencia son una parte importante de los elementos que debe conocer un ingeniero, estos procesos son los que nos permiten saber cómo acceden los usuarios a las diferentes tablas que compone una base de datos y poder evaluar su rendimiento y las posibilidades que existan para que ocurran errores.

Conocer su funcionamiento y las posibilidades de manejo serán de mucha utilidad para los estudiantes, que ante un proceso que realmente se maneja debajo de la mesa, puedan tener los criterios y conocimientos suficientes para determinar las causas y las soluciones de los diversos problemas que eventualmente se pueden presentar.

Todo este proceso será descrito de manera explícita en la siguiente cartilla dando ejemplos claros de los eventos que se pueden presentar y cuál sería la mejor manera de resolverlos.

Cada una de los procesos que contemplaremos en este módulo será descrito de manera clara y nos permitirá tomar acciones dependido de la problemática, con el fin de resolver las diferentes inquietudes y además generar coherencia dentro de los resultados que se obtendrán en el aprendizaje y manejo por parte de los estudiantes.

U3 Metodología

Con el fin de que el estudiante realice la mayor aprensión del conocimiento se realizaran las siguientes recomendaciones metodológicas:

Realizar las lecturas complementarias las cuales le permitirán ampliar conceptos y comprender la temática tratada en la Unidad

Utilizar fuentes bibliográficas e información de internet, recolectada para una mayor comprensión de la información sobre los temas propuestos.

Realizar prácticas reales con datos propios los cuales les permitirán corroborar y experimentar el funcionamiento y las diferentes maneras que hay para trabajar las múltiples funcionalidades que tiene el lenguaje SQL y además también conocer las excepciones y reglas que mantiene la integridad de los datos.

U3

Desarrollo temático

Control de concurrencia

Definición

La concurrencia es un fenómeno que se presenta en varios contextos. Uno de ellos es la multiprogramación ya que el tiempo del procesador es compartido dinámicamente por varios procesos. Otro caso son las aplicaciones estructuradas, donde la programación estructurada se implementa como un conjunto de procesos concurrentes. Y por último se tiene que la misma estructuración recién mencionada es utilizada en el diseño de los sistemas operativos, los cuales se implementan como un conjunto de procesos.

El término concurrencia se refiere al hecho de que los Manejadores de Bases de Datos permiten que muchas transacciones puedan accesar a una misma base de datos al mismo tiempo.

En un sistema de estos se necesitan algún tipo de mecanismos de control de concurrencia para asegurar que las transacciones concurrentes no interfieran entre sí.

En sistemas multiusuario, es necesario un mecanismo para controlar la concurrencia. Se pueden producir inconsistencias importantes, en la información derivadas del acceso concurrente, como por ejemplo, el problema de la operación perdida.

En un sistema de biblioteca, existe un campo que almacena el número de copias disponibles para préstamo. Este campo debe incrementarse en uno cada vez que se devuelve un ejemplar del libro y disminuirse en uno cada vez que se presta un ejemplar.

Si existen varias bibliotecarias, una de ellas inicia la transacción t1, leyendo la variable numero ejemplares (n), cuyo contenido se guarda en la variable n1. Tiempo después, otra bibliotecaria podría leer la misma variable incrementándola en una unidad, transacción t2. Después, la transacción t1 añade una unidad a esa variable y la actualiza, el resultado es erróneo, ya que la variable N debería haber aumentado en 2 unidades, y solo ha aumentado en una. La transacción t2 se ha perdido.

Consiste en controlar la interacción entre los usuarios concurrentes para no afectar la inconsistencia de los datos.

Cuando se trabajaba con manejadores de archivos y se estaba actualizando un determinado registro, ningún otro usuario podía actualizar el mismo archivo; aunque el registro a trabajar fuera otro. Con esta característica, ya más de un usuario puede accesar a un mismo registro y actualizarlo.

El objetivo fundamental del control de concurrencia de base de datos es garantizar que

la ejecución concurrente de transacciones no resulta en una pérdida de consistencia de la base de datos.

La concurrencia se da cuando dos transacciones están interconectadas, lo que es común en un entorno multiusuario. Así pues en un entorno de multiprogramación es posible ejecutar varias transacciones de manera concurrente.

Problemas que se presentan

Modificación perdida

En T1 (Tiempo 1), arranca TA (Transacción A), leen dato "X"

En T2 (Tiempo 2), arranca TB (Transacción B), leen dato "X" datos = 100

En T3 (Tiempo 3), modifica TA (Transacción A), dato "X" (aumenta el 100%) datos = 200

En T4 (Tiempo 4), modifica TB, dato "X" (en base a lo que leyó en T2) (aumenta el 50%) 150 datos final.

Dependencia no COMMITADA

Permitir leer un dato sin esperar que una transacción que la estaba modificando haga su Commit.

Análisis consistente

TA (Transacción A): suma saldos

TB (Transacción B): transfiere $10 de cuenta 1 a cuenta3

CUENTA 1	50
CUENTA 2	40
CUENTA 3	30

TA CUENTA 1 = $50
T1 SALDO = $50

_____

TA CUENTA 2 = $40
T2 SALDO 2 = $90

_____

TB CUENTA 1 = $50
T3 CUENTA 1 = $50 - $10
CUENTA 1 = $40

_____

TB CUENTA 3 = $30 + $10

T4 CUENTA 3 = $40

_____

TA CUENTA 3 = $40

T5 SALDO = $130 (*En realidad es $ 120*)

Para eliminar o disminuir estos 3 problemas se utilizan protocolos:

Bloqueos

- S compartido (para lecturas)
- X exclusivo, este puede ser por páginas/ tabla/ registros

TB/ TA	NO BLOQUEO	S	X
No bloqueo	Accede a objeto	Accede a objeto	No accede
S	Accede a objeto	Accede a objeto	No accede
X	No accede	No accede	No accede

Cuadro 1
Fuente: Propia.

Problemas con los bloqueos exclusivos: puede haber un bloqueo mortal o Deadlock.

TA	TB
I	J
J	I

Cuadro 2
Fuente: Propia.

Recursos

Políticas para terminar con el bloqueo mortal

- Matar los dos.
- Darles un tiempo de vida.
- Matar a la más vieja.
- Matar al azar.

Existen tres formas en que una transacción, aunque sea correcta, puede producir una respuesta incorrecta si alguna otra transacción interfiere con ella en alguna forma.

Observe que la transacción que interfiere también puede ser correcta por si misma; lo que produce el resultado incorrecto general, es el intercalado sin control de las operaciones de las 2 transacciones correctas.

Actualización perdida

Este problema puede presentarse si dos transacciones concurrentes actualizan una misma tupla, y la segunda que se realiza al final, no toma en cuenta el efecto de la primera.

TRANSACCION A TIEMPO TRANSACCION B
Recuperar T T1
T2 recuperar T
Actualizar T T3
T4 Actualizar T

Dependencia no confirmada

Ocurre si se permite que una transacción lea o modifique una tupla que ha sido modificada por otra transacción sin que se haya comandado el registro en firme de esta modificación, si ocurriera una operación de cancelación podrían generarse inconsistencias.

TRANSACCION A TIEMPO TRANSACCION B
T1 actualizar T
Recuperar T T2
T3 ROLLBACK

(La transacción A llega a ser dependiente de un cambio no confirmado en el tiempo T2)

TRANSACCION A TIEMPO TRANSACCION B
T1 actualizar T
Actualizar T T2
T3 ROLLBACK

(La transacción A actualiza un cambio no confirmado en el tiempo T2 y pierde esa actualización en el tiempo T3)

Análisis inconsistente

Ocurre cuando una transacción hace un análisis contable o estadístico, sobre una tupla que está siendo actualizada por otra transacción.

TRANSACCION A TIEMPO TRANSACCION B
Recuperar ACC1 T1
Suma = 40
Recuperar ACC2 T2
Suma = 90
T3 recuperar ACC3
T4 actualizar ACC3
30 20
T5 recuperar ACC1
T6 actualizar ACC1
50
T7 COMMIT
Recuperar ACC3 T8
Suma: 110 y no 120
TRANSACCION A
Suma saldos de cuenta ACC1: 40 50
ACC2: 50
TRANSACCION B
Transfiere una cantidad de 10
De la cuenta 3 a la 1 ACC3: 30 20

En los sistemas de tiempo compartido se presentan muchos problemas debido a que los procesos compiten por los recursos del sistema. Los programas concurrentes a diferencia de los programas secuenciales tienen una serie de problemas muy particulares derivados de las características de la concurrencia:

Violación de la exclusión mutua: en ocasiones ciertas acciones que se realizan en un programa concurrente no proporcionan los resultados deseados. Esto se debe a que existe una parte del programa donde se realizan dichas acciones que constituye una región critica, es decir, es una parte del programa en la que se debe garantizar que si un proceso accede a la misma, ningún otro podrá acceder. Se necesita pues garantizar la exclusión mutua.

Bloqueo mutuo o Deadlock: un proceso se encuentra en estado de Deadlock si está esperando por un suceso que no ocurrirá nunca. Se puede producir en la comunicación de procesos y más frecuentemente en la gestión de recursos. Existen cuatro condiciones necesarias para que se pueda producir Deadlock:

- Los procesos necesitan acceso exclusivo a los recursos.
- Los procesos necesitan mantener ciertos recursos exclusivos mientras esperan por otros.
- Los recursos no se pueden obtener de los procesos que están a la espera.
- Existe una cadena circular de procesos en la cual cada proceso posee uno o más de los recursos que necesita el siguiente proceso en la cadena.

Retraso indefinido o starvation: un proceso se encuentra en starvation si es retrasado indefinidamente esperando un suceso que no puede ocurrir. Esta situación se puede producir si la gestión de recursos emplea un algoritmo en el que no se tenga en cuenta el tiempo de espera del proceso.

Injusticia o unfairness: se pueden dar situaciones en las que exista cierta injusticia en relación a la evolución de un proceso. Se deben evitar situaciones de tal forma que se garantice que un proceso evoluciona y satisface sus necesidades sucesivas en algún momento.

Espera ocupada: en ocasiones cuando un proceso se encuentra a la espera por un suceso, una forma de comprobar si el suceso se ha producido es verificando continuamente si el mismo se ha realizado ya. Esta solución de espera es muy poco efectiva, porque desperdicia tiempo de procesamiento, y se debe evitar. La solución ideal es suspender el proceso y continuar cuando se haya cumplido la condición de espera.

Condiciones de carrera o competencia: la condición de carrera (race condition) ocurre cuando dos o más procesos acceden un recurso compartido sin control, de manera que el resultado combinado de este acceso depende del orden de llegada.

Postergación o aplazamiento indefinido(a): consiste en el hecho de que uno o varios procesos nunca reciban el suficiente tiempo de ejecución para terminar a su tarea. Por ejemplo, que un proceso ocupa un recurso y lo marque como ocupado y que termine sin marcarlo como desocupado. Si algún otro proceso pide ese recurso, lo vera ocupado y esperara indefinidamente a que se desocupe.

Condición de espera circular: esto ocurre cuando dos o más procesos forman una cadena de espera que los involucra a todos.

Condición de no apropiación: esta condición no resulta precisamente de la concurrencia, pero juega un papel muy importante en este ambiente. Esta condición especifica que si un proceso tiene asignado un recurso, dicho recurso no puede arrebatársele por ningún motivo, y estará disponible hasta que el proceso lo suelte por su voluntad.

Control de concurrencia en bases de datos relacionales

La mayoría de las bases de datos se utilizan en entornos multiusuario, en los que muchos clientes utilizando la misma aplicación, o muchas aplicaciones cada una con uno o muchos clientes acceden a la misma base de datos. Cada una de esas aplicaciones enviará consultas al gestor, y normalmente cada hilo de ejecución será una transacción diferente.

En la mayoría de los sistemas operativos actuales, las diferentes tareas o hilos se ejecutan de forma intercalada. Es decir, el sistema operativo decide por su cuenta cuando suspender una de las tareas y darle un poco de tiempo de ejecución a otra. Si hay tareas simultáneas o concurrentes sobre la misma base de datos, esta intercalación puede resultar en que las lecturas y escrituras de las diferentes tareas o aplicaciones en el medio físico se realicen en cualquier orden y secuencia.

El acceso simultáneo descrito puede dar como resultados información inconsistente o simplemente incorrecta, dependiendo de la mala o buena suerte que tengamos en la intercalación de las lecturas y escrituras simultáneas. Esta problemática ha llevado a diseñar e implementar diferentes estrategias de **control de concurrencia**, que se encargan de evitar todos esos problemas, de modo que los desarrolladores de las aplicaciones pueden "olvidarse" de ellos al escribir su código.

Por ejemplo, si tenemos una estructura de tablas relacional que incluye las siguientes:

PEDIDO (id, numcliente, idprod, cantidad, precio).
PRODUCTO (idprod, nombre, ..., stock).

Pueden ocurrir diferentes problemas relacionados con la escritura simultánea con otras escrituras o lecturas, incluyendo los siguientes:

1. Dos sentencias UPDATE que actualicen un mismo producto disminuyendo el stock del mismo en una unidad podrían terminar en que una de ellas no se realizase. Si pensamos en un UPDATE como una secuencia de una lectura y una escritura, puede que ambos UPDATE hagan la lectura, por ejemplo, de un stock de 10, y después las escrituras, disminuyen ese dato, quedando el resultado en 9, mientras que lo correcto era un resultado de 8.
2. Supongamos una sentencia que primero comprueba que hay stock del producto P, y después inserta un nuevo PEDIDO de diez unidades del producto P, que tiene un stock de 10, seguido de un UPDATE al stock por esa cantidad. Puede que otra inserción de un pedido se ejecute antes del UPDATE pero después de la comprobación, haciendo quedar el stock del producto en negativo.

Existen varias técnicas para controlar la concurrencia. Los bloqueos son los más conocidos, aunque también se utiliza el control de versiones múltiples y otras técnicas como las marcas de tiempo.

Los bloqueos como solución al problema de la concurrencia

Una forma de controlar la concurrencia es hacer que cada transacción deba adquirir un derecho de acceso exclusivo a cada fragmento de datos que necesite modificar. A estos "derechos" se les denomina bloqueos.

Bloqueos binarios

La forma más simple de bloquear es utilizar bloqueos binarios. En un bloqueo binario, cada transacción debe solicitar el bloqueo de cada fragmento de datos A que vaya a utilizar antes de acceder a él (sea para leerlo o escribirlo), mediante una operación bloquear(A). Deberá liberar todos los bloqueos, mediante una operación desbloquear(A) de modo que otras tareas puedan tomarlos.

Este sistema de bloqueos tiene una implementación muy simple, ya que solo requiere mantener una tabla que indica qué partes de los datos está bloqueada y por qué transacción.

Bloqueos de lectura/escritura

El sistema de bloqueos binarios es simple pero demasiado restrictivo, ya que no permite que dos transacciones que van a leer el mismo fragmento de datos A lo hagan simultáneamente, cuando en realidad, no puede haber problemas en varios lectores simultáneos. Los bloqueos de lectura/escritura hacen más débil la restricción permitiendo la siguiente compatibilidad de bloqueos.

	LECTURA	ESCRITURA
LECTURA	Compatible	Incompatible
ESCRITURA	Incompatible	Incompatible

Cuadro 3
Fuente: Propia.

En este caso, las operaciones que las transacciones deben realizar son tres: desbloquear(A) y bloquearparalectura(A) o bloquearparaescritura(A).

Estas llamadas se implementan de diferentes maneras dependiendo el gestor de bases de datos. En MySQL, tanto las solicitudes de bloqueos como las liberaciones se realizan mediante una sola llamada del API de los gestores de almacenamiento:

store_lock(THD *thd, THR_LOCK_DATA **to, enum thr_lock_type lock_type)

Cada llamada a store_lock utiliza el manejador de una tabla thd concreta, y se indica la información de los datos a bloquear mediante la variable to, y el tipo de bloqueo mediante lock_type (el número de tipos definidos en MySQL es muy grande, ya que cubre todos los tipos de bloqueo que implementan los múltiples gestores de almacenamiento disponibles).

Serialización de los bloqueos de lectura/escritura

La serialización de las operaciones de lectura y escritura consiste en ordenar esas operaciones para un conjunto de transacciones concurrentes de modo que los resultados de las operaciones sean correctos. Por ejemplo, si tenemos las siguientes transacciones X e Y, puede darse la siguiente situación.

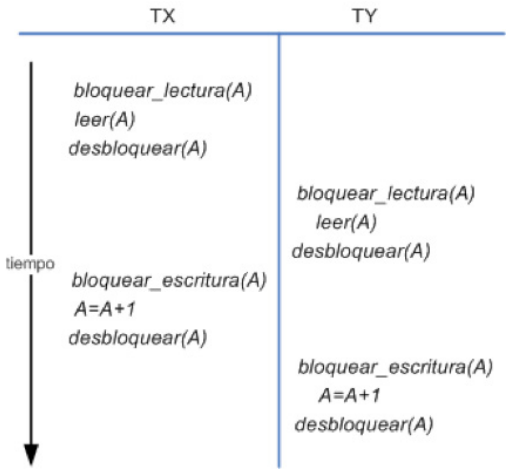


Imagen 1

Fuente: http://cnx.org/resources/118900b65e70792c295c2d9cabb618bfee06b9ad/graphics2.png

En esa situación, la ejecución de TX y TY hace que el dato original sólo se haya incrementado una vez, cuando tenía que haberse incrementado dos veces. Sin embargo, si hubiésemos tenido suerte y todas las operaciones de TX se hubiesen realizado antes que las de TY, el resultado habría sido correcto.

La conclusión es que el solo mecanismo de los bloqueos garantiza el acceso exclusivo a un dato o fragmento de información evitando algunos problemas, pero los problemas asociados a la intercalación de las operaciones compuestas aún pueden darse.

Por lo tanto, hace falta alguna política o mecanismo de adquisición y liberación de bloqueos que permita hacer las operaciones serializables.

El bloqueo en dos fases permite la serialización

El protocolo de bloqueo en dos fases fuerza a las transacciones cuando todas las operaciones de adquisición de bloqueos (bloquear_lectura, bloquear_escritura) preceden a la primera operación de desbloqueo (desbloquear). Dicho de otro modo, primero hay que adquirir todos los bloqueos, y después se pueden liberar.

Cuando se utiliza el protocolo de bloqueo en dos fases, puede demostrarse que la ejecución será serializable.

Inconvenientes de los bloqueos y la serialización

Un problema del protocolo de bloqueo en dos fases es que puede llevar a situaciones de interbloqueo. La siguiente es una secuencia de operaciones que lleva al interbloqueo pero cumple perfectamente l protocolo de bloqueo en dos fases.

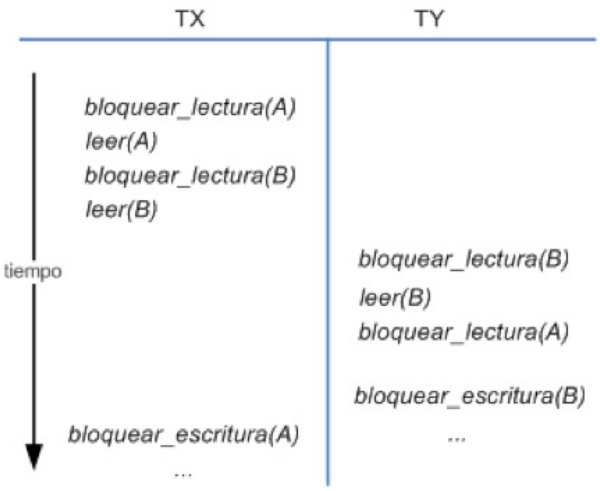


Imagen 2
Fuente: http://cnx.org/resources/118900b65e70792c295c2d9cabb618bfee06b9ad/graphics2.png

El inconveniente está en que puede que en la fase de adquisición de bloqueos (fase de expansión), más de una transacción esté interesada en los mismos dos fragmentos de datos, y la mala suerte nos lleve a una situación en la que ambos quedan suspendidos, sin posibilidad de avanzar.

¿Bloqueos más grandes o más pequeños?

Un aspecto que aún no se ha tratado en la discusión anterior es cuán "grandes" son los elementos de datos que se deben bloquear. Una opción posible es bloquear tablas enteras. Esto hace que la gestión de los bloqueos sea simple y tenga poca sobrecarga (solo hay que guardar el estado de bloqueo de las N tablas). No obstante, esto impide que dos transacciones que van a manipular filas diferentes de una tabla puedan progresar en paralelo.

Una segunda opción es utilizar bloqueos al nivel de las filas. En este caso, se hacen mayores las posibilidades de concurrencia, pero por otro lado, hay que mantener mucha más información sobre los bloqueos (ya que el número de filas es en general muy grande respecto al número de tablas), y el servidor se sobrecarga más con la gestión de los bloqueos.

Hay gestores de bases de datos que permiten seleccionar el tipo de bloqueo que queremos para nuestra base de datos. Por ejemplo, en MySQL hay gestores de almacenamiento que ofrecen bloqueo a nivel de fila, y otros bloqueos a nivel de tabla.

No obstante lo anterior, hay sentencias que siempre producirán un bloqueo de tabla, como por ejemplo una sentencia ALTER TABLE.

Guardando "instantáneas" de los datos: el Control Multi-versión

El protocolo de bloqueo en dos fases limita considerablemente las posibilidades de concurrencia. Si observamos los problemas que causan los bloqueos no serializados, veremos que muchos de los problemas están en que una transacción lee un cierto dato y antes de escribir el resultado, otra transacción lee el dato "antiguo". En ese momento, cada transacción trabaja con un estado de información inconsistente.

Para paliar esos problemas, pero permitir la mayor concurrencia posible, se han diseñado los protocolos de control multi-versión.

La idea básica es que cuando una transacción modifica un dato, se crea una nueva versión del mismo, pero se guarda la anterior. De este modo, al acabar la ejecución de las transacciones, se puede utilizar para cada una de ellas la versión de los datos "que hace la ejecución correcta", es decir, que hace la ejecución serializable.

Protocolos

Protocolo de bloqueo de dos fases

Protocolo que asegura la secuencialidad es el protocolo de bloqueo de dos fases. Este protocolo exige que cada transacción realice las peticiones de bloqueo y desbloqueo de dos fases.

1. Fase de crecimiento.- Una transacción puede obtener bloqueos pero no puede liberarlos.
2. Fase de decrecimiento.- Una transacción puede liberar bloqueos pero no puede obtener ninguno nuevo.

Inicialmente una transacción está en la fase de crecimiento. La transacción adquiere los bloqueos que necesite. Una vez que la transacción entra un bloqueo, entra en la fase de crecimiento como puede alcanzar peticiones de bloqueo.

Se puede mostrar que el protocolo de bloqueo de dos fases asegura secuencialidad en cuanto a conflictos. Considérese cualquier transacción. El punto de la planificación en el cual la transacción obtiene el bloqueo final (el final de la fase de crecimiento) se denomina punto de bloqueo de la transacción.

El protocolo de bloqueo de dos fases no asegura la ausencia de interbloqueos.

Los retrocesos en cascada se pueden evitar por medio de una modificación del protocolo de dos fases que se denomina protocolo de bloque estricto de dos fases.

El *protocolo de bloque estricto de dos fases* exige que además de que el bloqueo sea de dos fases, una transacción debe poseer todos los bloqueos en modo exclusivo que tome hasta que dicha transacción no comprometida está bloqueado en modo exclusivo hasta que la transacción lea el dato.

Otra variante del bloqueo de dos fases es el protocolo de bloqueo riguroso de dos fases, el cual exige que se posean todos los bloqueos hasta que se comprometan la transacción. Se puede comprobar fácilmente que con el bloqueo riguroso de dos fases se pueden secuenciar las transacciones en el orden en que se comprometen.

Muchos sistemas de base de datos implementan tanto el bloqueo estricto como el bloqueo estricto de dos fases.

Protocolos basados en grafos

En ausencia de información acerca de la forma en que se accede a los elementos de datos, el protocolo de bloqueo de dos fases es necesario y suficiente para asegurar la secuencialidad. Así, si se desea desarrollar protocolos que no sean de dos fases, es necesario tener información adicional acerca de la forma en que cada transacción accede a la base de datos.

Existen varios modelos que difieren sea en la cantidad de información que se proporciona. El modelo más simple que tenga conocimiento previo acerca con el cual accede a los elementos de la base de datos.

Dada la información es posible construir protocolos de bloqueos que no sean de dos fases pero que un obstante persigue la secuencialidad para adquirir al conocimiento previo que impone orden para el conjunto D={d1,.....,dn}.

El orden parcial implica que el conjunto D se puede ver como un grafo aciclico dirigido grafo de la base de datos. En este apartado, para simplificar, se centra la atención solo en aquellos grafos que sean árboles con raíz. Se puede presentar un protocolo simple llamado protocolo de árbol, el cual está restringido a utilizar solo bloqueos de archivos.

Para ilustrar este protocolo, considérese el grafo de la base de datos de la Figura siguiente.

Las 4 transacciones siguientes siguen el protocolo de árbol sobre dicho grafo. Solo de muestran las instrucciones de bloqueo y desbloqueo:

T10: bloquear-X(B) ;bloquear-X(E); bloquear-X(D); desbloquear(B) ; desbloquear. (E); bloquear-X(G); desbloquear (D); desbloquear (G).
T11: bloquear-X(D) ;bloquear-X(H); desbloquear(D) ; desbloquear (H).
T12: bloquear-X(B) ;bloquear-X(E); desbloquear (E); desbloquear (B).
T13: bloquear-X(D); bloquear-X(H); desbloquear (D); desbloquear (H).

- Si se quiere conseguir mayor grado de paralelismo hay que utilizar otro tipo de protocolos.
- Los protocolos que no son del tipo anterior necesitan información adicional.
- Un modelo sencillo se basa en conocer el orden en que se accede a la información.
- Si se puede establecer un orden parcial en el acceso a los datos, se puede aplicar un esquema de cerrojos mucho más eficiente.
- El orden parcial se puede representar en un grafo.
- Los protocolos en árbol garantizan la seriabilidad y la ausencia de bloqueos.
- La liberación del cerrojo se puede producir antes en este protocolo que en el de bloqueo en dos fases.
  - Tiempos de espera más reducidos
  - aumento de la concurrencia
  - ausencia de bloqueos
- El inconveniente es que puede tener que bloquear datos que no necesita manejar.
  - aumenta el bloqueo, mayor tiempo de espera.
  - potencial reducción de concurrencia.

Funcionamiento: se distinguen dos fases: votación y decisión.

En la primera el coordinador pregunta si están preparados para realizar (Commit) la transacción. Si un participante vota abortar, o no responde dentro del timeout, entonces se aborta la transacción. Si todos votan para realizarla (Commit), entonces el coordinador da instrucciones para realizar la transacción.

El periodo de timeout permite evitar el bloqueo de los nodos ante el fallo de uno de los participantes. Si un nodo falla en el proceso, todos siguen un protocolo de terminación, el nodo fallido sigue un protocolo de recuperación.

Protocolo en árbol

- Los nodos representan los datos.
- Los ejes el orden en que se acceden.
- Es el protocolo de grafo más sencillo
- Utiliza solo cerrojos exclusivos.

1. El primer cerrojo de una transacción puede ser sobre cualquier dato.
2. Los siguientes datos se pueden bloquear si el padre del dato está bloqueado por la misma transacción.
3. Los datos se pueden desbloquear en cualquier momento.
4. Un dato que ha sido bloqueado y desbloqueado, no puede ser bloqueado de nuevo.

Protocolos basados en marcas temporales

Otro método para determinar el orden de secuencialidad es seleccionar previamente un orden entre las transacciones en el método más común para hacer esto es utilizar un esquema de ordenación por marcas temporales.

Protocolo de ordenación por marcas temporales

El protocolo de ordenación por marcas temporales asegura que todas las operaciones leer y escribir conflictivas se ejecutan en el orden de las marcas temporales.

El protocolo de ordenación por marcas temporales asegura la secuencialidad en cuantos conflictos. Esta afirmación se deduce del hecho de que las operaciones conflictivas se procesan durante la ordenación de las marcas temporales. El protocolo asegura la ausencia de interbloqueos, ya que ninguna transacción tiene que esperar. El protocolo puede generar planificaciones no recuperables; sin embargo se puede extender para producir planificaciones sin cascada.

Marcas de tiempo multiversión

El mecanismo de marcas de tiempo supone que existe una única versión de los datos; por lo que sólo una transacción puede acceder a los mismos. Se puede relajar esta restricción permitiendo que varias transacciones lean y escriban diferentes versiones del mismo dato siempre que cada transacción vea un conjunto consistente de versiones de todos los datos a los que accede.

El problema con las técnicas de bloqueo es que puede producirse un interbloqueo (llamado Deadlock), que es una situación que se da cuando dos o más transacciones están esperando cada una de ellas que otra libere algún objeto antes de seguir. Este problema, que ha sido estudiado en profundidad en los sistemas operativos, puede tener dos soluciones:

- Prevenir el interbloqueo, obligando a que las transacciones bloqueen todos los elementos que necesitan por adelantado.
- Detectar el interbloqueo, controlando de forma periódica si se ha producido, para lo que suele construirse un grafo de espera. Si existe un ciclo en el grafo se tiene un interbloqueo. Cada SGBD tiene su propia política para escoger las víctimas, aunque suelen ser las transacciones más recientes.

Protocolos basados en validación

Para aquellos casos en los que la mayoría de las transacciones son de solo lectura. La tasa de conflictos entre las transacciones puede ser baja.

Así muchas de esas transacciones, si se ejecutasen sin la supervisión en un sistema de control de concurrencia, llevarían no obstante al sistema a un estado consistente.

Un esquema de control de concurrencia supone una sobrecarga en la ejecución del código y un posible retardo en las transacciones. La dificultad de reducir la sobrecarga está en que no se conoce de antema las transacciones que estarán involucradas en unos conflictos. Para obtener dicho conocimiento se necesita un esquema para observar el sistema.

Se asume que cada transacción Ti se ejecuta en dos o tres fases diferentes durante su tiempo de vida, dependiendo de sí es una transacción de solo lectura o una actualización. Las fases son en orden que sigue:

1. Fase de lectura.- Durante esta fase tiene lugar la ejecución dela transacción Ti. Todas las operaciones escribir se realizan sobre las variables locales temporales sin actualizar la base de datos actual.
2. Fase de validación.-La transacción Ti realiza una prueba de validación para determinar si puede copiar a la base de datos las variables locales temporales que tienen como resultado de las operaciones escribir sin causar una violación de la secuencialidad.
3. Fase de escritura.- Si la transacción Ti tiene éxito en la validación entonces las actualizaciones reales se aplican a la base de datos. En otro caso se retrocede.

Protocolos basados en cerrojos

- El aislamiento en las transacciones viene impuesto por el acceso a los datos.
- Una forma de garantizar la serialización es garantizar el acceso exclusivo a los datos.
- Definiendo regiones críticas específicas de un dato.
- El método más utilizado es manejar el acceso a los datos a través de cerrojos.

Protocolo de cerrojo en dos fases

- Asegura planes de ejecución seriables.
- Protocolo en dos fases.

Fase 1: Agrupamiento

- la transacción puede obtener cerrojos.
- la transacción no puede liberar cerrojos.

Fase 2: Reducción

- la transacción puede liberar recursos.
- la transacción no puede obtener cerrojos.

- Se puede probar que las transacciones son seriables en el orden que apuntan los cerrojos (donde se adquiere el cerrojo final).
- Este protocolo no evita los bloqueos.

Protocolos basados en bloqueo

Una forma de asegurar la secuencialidad es exigir que el acceso a los elementos de datos se haga en exclusión mutua, es decir, mientras una transacción accede a un elemento de datos, ninguna otra transacción puede modificar dicho elemento. El método más habitual que se usa para implementar

este requisito es permitir que una transacción acceda a un elemento de datos sólo si posee actualmente un bloqueo sobre dicho elemento.

Existen varios protocolos basados en marcas de tiempo, entre los que destacan:

WAIT-DIE que fuerza a una transacción a esperar en caso de que entre en conflicto con otra transacción cuya marca de tiempo sea más reciente, o a morir (abortar y reiniciar) si la transacción que se está ejecutando es más antigua.

WOUND-WAIT, que permite a una transacción matar a otra que posea una marca de tiempo más reciente, o que fuerza a la transacción peticionaria a esperar.

Estos son todos de dos fases:

- 2PL centralizado.
- 2PL con copia primaria.
- 2PL distribuido.
- bloque mayoritario.

2PL centralizado. Se caracteriza por:

Hay un único planificador, o gestor de bloqueos (lock manager), para la totalidad del SGBD Distribuido que pueden garantizar (grant) y liberar (release) bloqueos.

2PL centralizado. Funcionamiento:

El coordinador de transacciones local divide la transacción en subtransacciones, usando el catálogo global del sistema. Si la transacción implica actualizar un dato que está replicado, el coordinador solicita un bloqueo de escritura de todas las copias antes de actualizar cada copia y liberar los bloqueos. El coordinador puede elegir cualquiera de las copias de un dato replicado para lectura.

El gestor de transacciones local, implicado en la transacción global, solicita y libera los bloqueos que mantiene el gestor centralizado de bloqueos usando las reglas usuales para el bloqueo en dos fases.

El gestor centralizado de bloqueos comprueba que las peticiones de bloqueo sobre un dato sean compatibles, de manera que el gestor de bloqueos envía un mensaje de vuelta al nodo que originó la petición reconociendo que el bloqueo ha sido concedido. En caso contrario, coloca la petición en una cola hasta que el bloqueo pueda ser realizado.

2PL con copia primaria. Se caracteriza por:

Cada gestor de bloqueos local es responsable de un conjunto de datos, de manera que se elige una copia como copia primaria; el resto de copia se llaman copias esclavas. La elección del nodo primario es flexible, y el nodo elegido no contiene necesariamente la copia primaria de ese dato.

2PL distribuido Se caracteriza por:

Se distribuye un gestor de bloqueo en cada nodo. Cada uno es responsable de la gestión de bloqueos de los datos que contiene en ese nodo. El 2PL distribuido implementa un protocolo de control de réplicas Read-One-Write-All.

Cualquier copia de un dato replicado puede ser usada para operaciones de lectura, pero todas las copias deben ser bloqueadas para escritura antes que se puedan modificar.

Bloqueo mayoritario. Se caracteriza por:

Se mantiene un gestor de bloqueos en cada nodo para gestionar los bloqueos de ese nodo. Cuando se ejecuta una transacción

que trabaja con un dato que esta replicado, debe enviar una petición de bloqueo a más de la mitad de los nodos donde está el dato.

Protocolo de recuperación. Fallo en el coordinador

Estado INICIAL: la recuperación en este caso consiste en iniciar el procedimiento de ejecución (Commit).

Estado de ESPERA: el coordinador no ha recibido todas las respuestas (ninguna de abortar). La recuperación consiste en reiniciar el procedimiento de ejecución (Commit).

Estado DECIDIDO: el coordinador ha dado instrucciones para abortar o ejecutar globalmente la transacción. Al reiniciar, si el coordinador ha recibido todos los reconocimientos, completa la transacción con éxito, en caso contrario, tiene que iniciar el protocolo de terminación.

Protocolo de recuperación. Fallo en un participante:

Estado INICIAL: el participante no ha votado todavía sobre la transacción, de manera que cuando se recupera puede abortar.

Estado PREPARADO: el participante ha enviado su voto al coordinador. En este caso, la recuperación se hace mediante el protocolo de terminación discutido anteriormente.

Fallo en los estados ABORTADO/EJECUTADO: el participante ha completado la transacción. Por consiguiente, al reiniciar, ninguna otra acción será necesaria.

Este protocolo debe intentar que todos los participantes realicen las mismas acciones (estado consistente).

Protocolo de elección:

El protocolo es ejecutado cuando los participantes detectan que el coordinador ha fallado. Una asunción que hace este protocolo es que los nodos tienen preestablecido un orden.

Funcionamiento: cada nodo envía su número de orden al resto de nodos mayores que él. Si este recibe un mensaje de un nodo menor que el deja de enviar mensajes.

Deadlock

El Deadlock es una condición que ningún sistema o conjunto de procesos quisiera exhibir, ya que consiste en que se presentan al mismo tiempo cuatro condiciones necesarias: La condición de no apropiación, la condición de espera circular, la condición de exclusión mutua y la condición de ocupar y esperar un recurso. Ante esto, si el Deadlock involucra a todos los procesos del sistema, el sistema ya no podrá hacer algo productivo. Si este involucra algunos procesos, éstos quedarán congelados para siempre.

Causas del Dead Lock

Los puntos muertos ocurren cuando dos o más transacciones solicitan recursos en forma incremental y se bloquean mutuamente impidiéndose una a otra la conclusión. Las transacciones y recursos forman un ciclo.

Hasta es posible que un punto muerto se cree cuando se realiza al acceso a un solo objeto mediante dos transacciones y se aceptan reclamaciones increméntales. Este tipo de punto muerto tiene una mayor probabilidad de ocurrir si los objetos que se están asegurando son grandes cuando la unidad de aseguramiento es un archivo la primera reclamación puede emitirse para

lograr acceso a un registro y la reclamación incremental emitirse a fin de actualizar otro registro de archivo.

Puntos muertos debidos a recursos compartidos del sistema, los puntos muertos también pueden provocarse debido a la competencia por objetos que no estén identificados en forma específica, pero que sean miembros de una clase compartida de recursos. Un bloqueo temporal ocurre cuando los límites de un recurso se alcanzan.

Condiciones de punto muerto, la posibilidad de puntos muertos existen 4 condiciones:

- Seguros: la interferencia de acceso se resuelve posesionando y reputando los seguros.
- Bloqueo: un propietario de un objeto está bloqueado cuando solicita un objeto asegurado.
- Garantía de conclusión: los objetos no pueden quitarse de sus propietarios.
- Circulación: existe una secuencia circular de solicitud como se muestra en él.

En el área de la informática, el problema del Deadlock ha provocado y producido una serie de estudios y técnicas muy útiles, ya que éste puede surgir en una sola máquina o como consecuencia de compartir recursos en una red.

En el área de las bases de datos y sistemas distribuidos han surgido técnicas como el ‹two phase locking› y el ‹two phase Commit› que van más allá de este trabajo. Sin embargo, el interés principal sobre este problema se centra en generar técnicas para detectar, prevenir o corregir el Deadlock.

Las técnicas para prevenir el Deadlock consisten en proveer mecanismos para evitar que se presente una o varias de las cuatro condiciones necesarias del Deadlock. Algunas de ellas son:

- Asignar recursos en orden lineal: esto significa que todos los recursos están etiquetados con un valor diferente y los procesos solo pueden hacer peticiones de recursos ‹hacia adelante›. Esto es, que si un proceso tiene el recurso con etiqueta ‹5› no puede pedir recursos cuya etiqueta sea menor que ‹5›. Con esto se evita la condición de ocupar y esperar un recurso.
- Asignar todo o nada: este mecanismo consiste en que el proceso pida todos los recursos que va a necesitar de una vez y el sistema se los da solamente si puede dárselos todos, si no, no le da nada y lo bloquea.
- Algoritmo del banquero: este algoritmo usa una tabla de recursos para saber cuántos recursos tiene de todo tipo. También requiere que los procesos informen del máximo de recursos que va a usar de cada tipo. Cuando un proceso pide un recurso, el algoritmo verifica si asignándole ese recurso todavía le quedan otros del mismo tipo para que alguno de los procesos en el sistema todavía se le pueda dar hasta su máximo. Si la respuesta es afirmativa, el sistema se dice que está en ‹estado seguro› y se otorga el recurso. Si la respuesta es negativa, se dice que el sistema está en estado inseguro y se hace esperar a ese proceso.

Para detectar un Deadlock, se puede usar el mismo algoritmo del banquero, que aunque no dice que hay un Deadlock, sí dice cuándo se está en estado inseguro que es la antesala del Deadlock. Sin embargo, para detectarlo se pueden usar las ‹gráficas de recursos›.

En ellas se pueden usar cuadrados para indicar procesos y círculos para los recursos, y flechas para indicar si un recurso ya está asignado a un proceso o si un proceso está esperando un recurso. El Deadlock es detectado cuando se puede hacer un viaje de ida y vuelta desde un proceso o recurso. Por ejemplo, suponga los siguientes eventos:

Evento 1: Proceso A pide recurso 1 y se le asigna.
Evento 2: Proceso A termina su time slice.
Evento 3: Proceso B pide recurso 2 y se le asigna.
Evento 4: Proceso B termina su time slice.
Evento 5: Proceso C pide recurso 3 y se le asigna.
Evento 6: Proceso C pide recurso 1 y como lo está ocupando el proceso A, espera.
Evento 7: Proceso B pide recurso 3 y se bloquea porque lo ocupa el proceso C.
Evento 8: Proceso A pide recurso 2 y se bloquea porque lo ocupa el proceso B.

En la imagen 3 se observa como el ‹resource graph› fue evolucionando hasta que se presentó el Deadlock, el cual significa que se puede viajar por las flechas desde un proceso o recurso hasta regresar al punto de partida. En el Deadlock están involucrados los procesos A, B y C.

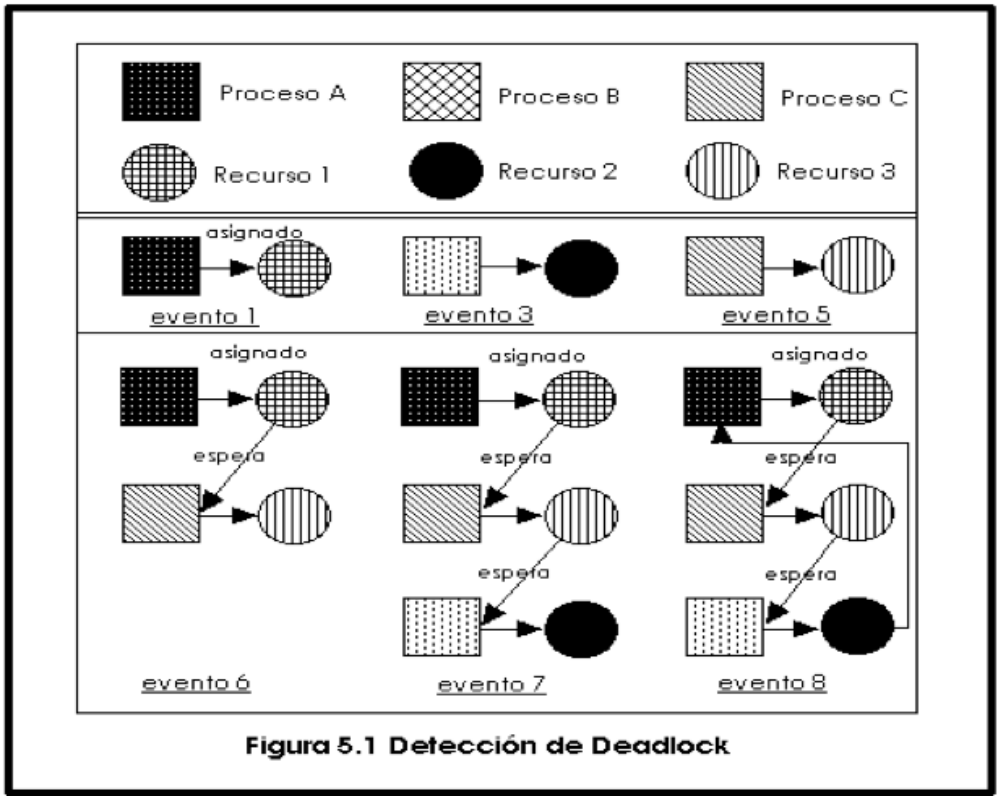


Figura 5.1 Detección de Deadlock

Imagen 3
Fuente: http://1.bp.blogspot.com/-YIgpOtBAlPc/T3p1ac70_ul/AAAAAAAAAHo/DBw5FqtDL2c/s400/000338632.png

Una vez que un Deadlock se detecta, es obvio que el sistema está en problemas y lo único que resta por hacer es una de dos cosas: tener algún mecanismo de suspensión o reanudación que permita copiar todo el contexto de un proceso incluyendo valores de memoria y aspecto de los periféricos que esté usando para reanudarlo otro día, o simplemente eliminar un proceso o arrebatarle el recurso, causando para ese proceso la pérdida de datos y tiempo.

Causas de punto muerto

Los puntos muertos ocurren cuando dos o más transacciones solicitan recursos en forma incremental y se bloquean mutuamente impidiéndose una a otra la conclusión. Las transacciones y recursos forman un ciclo.

Hasta es posible que un punto muerto se cree cuando se realiza al acceso a un solo objeto mediante dos transacciones y se aceptan reclamaciones increméntales. Este tipo de punto muerto tiene una mayor probabilidad de ocurrir si los objetos que se están asegurando son grandes cuando la unidad de aseguramiento es un archivo la primera reclamación puede emitirse para lograr acceso a un registro y la reclamación incremental emitirse a fin de actualizar otro registro de archivo.

Puntos muertos debidos a recursos compartidos del sistema, los puntos muertos también pueden provocarse debido a la competencia por objetos que no estén identificados en forma específica, pero que sean miembros de una clase compartida de recursos. Un bloqueo temporal ocurre cuando los límites de un recurso se alcanzan.

Las transacciones que se comunican a fin de procesar conjuntamente datos de proceso también están sujetas a puntos muertos, un ejemplo de transacción en cooperación de este tipo ocurre cuando los datos son trasladados entre archivos y buffers por una transacción y procesados simultáneamente por otras. Una transacción puede generar datos que se escriban en los archivos con una gran velocidad y apoderarse de todos los buffers. Ahora una transacción de análisis de datos no puede proceder debido a que la transacción que los trasladando puede entregar datos adicionales debido a escasez de buffers ya que la transacción de análisis de los datos no libera sus buffers hasta que concluya, de nuevo, se presenta una situación clásica de punto muerto, los mecanismos para manejar los puntos muertos deben, incluir, todos los objetos asegurables, esto podría incluir unidades de cinta, impresora y otros dispositivos, buffers y áreas de memoria asignados por el sistema operativo, archivos de bloques y registros controlados por el sistema de archivos y de la base de datos.

Condiciones de punto muerto, la posibilidad de puntos muertos existen cuatro condiciones [coffman]:

1. Seguros.- La interferencia de acceso se resuelve posesionando y reputando los seguros.
2. Bloqueo.- Un propietario de un objeto está bloqueado cuando solicita un objeto asegurado.
3. Garantía de conclusión.- Los objetos no pueden quitarse de sus propietarios.
4. Circularidad.- Existe una secuencia circular de solicitud.

Deadlocks distribuidos

En un ambiente distribuido la detección y recuperación es más compleja porque va-

rios nodos están involucrados y se requiere el intercambio de información entre los manejadores de la transacción distribuida. En general se postulan tres alternativas para manejar los Deadlocks.

- Detección de Deadlocks usando un control jerárquico centralizado.
- Detección distribuida del Deadlock.
- Prevención del Deadlock.

Como se puede apreciar, los métodos 1 y 2 son correctivos que necesariamente conllevan la pérdida de tiempo porque se tiene que abortar alguna transacción, el método 3 evita la aparición del Deadlock y teóricamente sería preferible implantarlo.

Técnicas para prevenirlo

Los métodos de corregir un Deadlock una vez que ya se presentó éste conllevan la pérdida de tiempo y no de información, ya que las transacciones puedes abortarse de manera segura debido a las facilidades de las bitácoras, el protocolo de bloqueo de dos fases y el protocolo Commit de dos fases.

El método de prevención del Deadlock también hace necesario abortar transacciones y siempre serán las transacciones más nuevas o jóvenes las abortadas para evitar los Deadlocks.

Existen dos métodos: el método apropiativo y el no apropiativo. Ambos se basan en la hora en que las transacciones son creadas, podemos hablar entonces de transacciones viejas y jóvenes.

En el método no apropiativo, si una transacción Ta pide un recurso bloqueado por Tb, se permite que Ta espere solamente si Ta es más vieja que Tb. Ta es reinicializada si es más joven que Tb, conservando siempre su antigüedad. Lo que se consigue con el método no apropiativo es que ninguna transacción joven esperará por las transacciones viejas, eliminando la condición de espera circular distribuida.

En el método apropiativo la regla es la inversa: Si Ta pide un bloqueo sobre un recurso cuyo dueño es Tb, se le permite esperar si Ta es más joven que Tb. Si Ta es más vieja, no espera sino que Tb es abortada por ser más joven y luego se reinicializa conservando su antigüedad.

También podemos anular alguna de las 4 condiciones necesarias para que se produzca un Deadlock:

No puede ser anulada porque existen recursos que deben ser usados en modalidad exclusiva.

La alternativa sería hacer que todos los procesos solicitaran todos los recursos que habrán de utilizar antes de utilizarlos al momento de su ejecución lo cual sería muy ineficiente. Para anular esta condición cuando un proceso solicita un recurso y este es negado el proceso deberá liberar sus recursos y solicitarlos nuevamente con los recursos adicionales. El problema es que hay recursos que no pueden ser interrumpidos.

Espera Circular: esta estrategia consiste en que el sistema operativo numere en forma exclusiva los recursos y obligue a los procesos a solicitar recursos en forma ascendente. El problema de esto es que quita posibilidades a la aplicación.

Técnicas para corregirlo

Si se tiene cuidado en la forma de asignar los recursos se pueden evitar situaciones de

Deadlock. Supongamos un ambiente en el que todos los procesos declaren a priori la cantidad máxima de recursos que había de usar. Estado Seguro: un estado es seguro si se pueden asignar recursos a cada proceso (hasta su máximo) en algún orden sin que se genere Deadlock. El estado es seguro si existe un ordenamiento de un conjunto de procesos {P1...Pn} tal que para cada Pi los recursos que Pi podrá utilizar pueden ser otorgados por los recursos disponibles más los recursos utilizados por los procesos Pj,j<i. Si los recursos solicitados por Pi no pueden ser otorgados, Pi espera a que todos los procesos Pj hayan terminado.

Como evitar los puntos muertos

Para evitar puntos muertos puede simplificar muchas de elecciones alternativas. Los esquemas para evitar estos casos imponen a los usuarios restricciones que pueden resultar difíciles de aceptar. Existen cuatro enfoques para puntos muertos:

1. Reparación posterior: no utilizar seguros y arreglar después las fallas por inconsistencia.
2. No bloquear: solamente afectar a quienes efectuaron solicitudes que provocaron reclamaciones en conflicto.
3. Asignación previa: si existe algún conflicto, quitar los objetos a sus propietarios.
4. Aseguramiento de dos fases: se realizan primero todas las reclamaciones y si ninguna está bloqueada se inician todas las modificaciones.

Reparación posterior

El primer enfoque, es reparar posteriormente los problemas debido a no asegurar, puede responder a un valido en sistemas experimentales y educativos, pero resulta inaceptable en la mayoría de las aplicaciones comerciales.

No bloquear

El segundo enfoque asigna la responsabilidad a la transacción, el sistema proporcionara un aviso de interferencia potencial al negar la solicitud de acceso explosivo.

Asignación previa

La asignación previa de reclamaciones otorgadas a las transacciones reúne de una capacidad de "volver a enrollar". La transacción, cuando se le notifica que no puede continuar, tiene que restaurar la base de datos y colocarse a sí misma en la línea de espera para un nuevo turno o el sistema tiene que eliminar la transacción, restaurar la base de datos y volver a iniciar de nuevo la transacción.

Secuencia previa

Consiste en evitar la circularidad en la secuencia de solicitud, en este caso existen 3 enfoques, vigilancia de la existencia para evitar la circularidad y aseguramiento de 2 fases, a fin de evitar puntos muertos puede vigilarse el patrón de solicitud para todas las transacciones.

Aseguramiento de dos fases

Un enfoque simple para evitar la circularidad consiste en hacer que se reclamen previo todos los objetos antes de otorgar ningún seguro, el reclamar los recursos antes de prometer otorgar el acceso a ellos significa que posiblemente una transacción no sea capaz de concluir la fase de reclamación previa. El problema de 2 fases es que la reclamación previa puede llegar a tener que reclamar más y mayores objetos de los que

en realidad se necesita, si un cálculo sobre los datos determina que objetos se necesitan después, es posible que se reclame en forma previa todo un archivo en vez de un registro.

Recursos reservados

Los puntos muertos provocados por la competencia de recursos clasificados, algunas veces se disminuye al no permitir que ninguna nueva transacción se inicie cuando la utilización llega al nivel. Esta técnica no asegura evitar los puntos muertos a menos que la reserva se conserve de un tamaño tan grande que resulte poco práctico, lo suficiente para permitir que todas las transacciones activas se concluyan.

Métodos de recuperación

Se usan dos métodos de recuperación de base de datos: recuperación en avance o recuperación en retroceso (roll-forward o roll-back).El método a utilizar depende del tipo y la extensión de los errores.

Recuperación en avance

Con este método se consigue la restauración mediante la copia de respaldo de la base para recuperar la porción principal de los datos. Después se reaplican los registros post-imagen del registro a la copia de respaldo para incorporar las actualizaciones efectuadas desde que se hizo la copia de respaldo. Los registros post-imagen del registro log, más que las transacciones, se reaplican ala copia de respaldo, ya que estas imágenes son datos procesados, listos para rescribirse en la base.

Recuperación en retroceso

Esta recuperación puede nulificar el efecto de una sola transacción que hizo cambios en la base pero abortó antes del término. (Recuerde que para mantener la integridad de los datos, una transacción debe ejecutarse en su totalidad o no ejecutarse). La recuperación en retroceso se consigue llamando al registro ante-imagen del archivo log y reinstalándolo en la base para nulificar el efecto de transacción errónea.

Operación de registro

La mayoría de los DBMS proporcionan un elemento de rastreo para registrar lo sucedido en cada transacción actualizada por la base de datos.

El registro, es un prerrequisito para la recuperación de la base de datos; restaura un archivo a su estado anterior cuando ocurre alguna falla en una transacción. El manejador del registro (log manager), es una componente de la DBMS que efectúa el registro escribiendo cualquiera de los dos tipos de registro siguiente en un archivo de búsqueda:

ante-imagen: se refiere a los bloques antiguos de datos originales en la base que se guardaron antes de la actualización de una transacción. Si ocurre un error, esta copia se puede usar para cancelar el efecto de la transacción.

Post-imagen: es el nombre que recibe un bloque procesado por una transacción listo para ser reescrito en la base.

Etiquetas de tiempo

Para evitar interferencia debida a la concurrencia se usa el aseguramiento o el tiempo de estampado para seriar la ejecución de transacciones concurrentes, de cualquier modo, estas estrategias requieren de complicados protocolos de control los que pue-

den deteriorar el desempeño del sistema al generar tráfico extra entre las distintas localidades.

Para establecer este ordenamiento, el administrador de transacciones le asigna a cada transacción T1 una estampa de tiempo única Ts (Ti) cuando esta inicia.

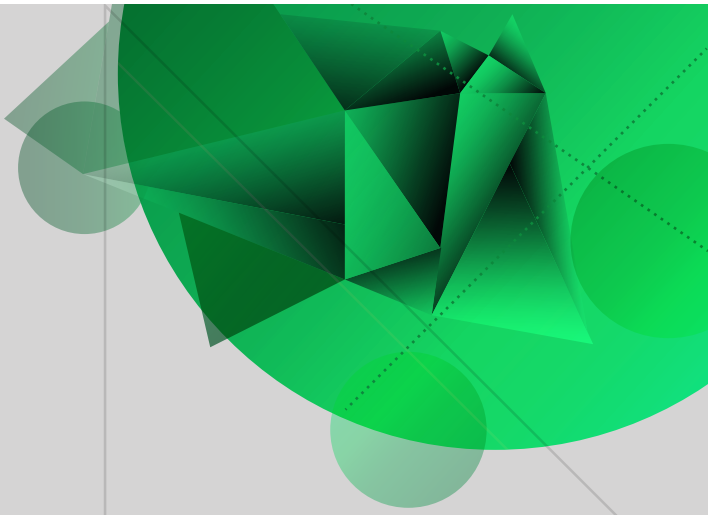
A diferencia de los algoritmos basados en candados, los algoritmos basados en estampas de tiempo no pretenden mantener la seriabilidad por exclusión mutua. En lugar de eso, ellos seleccionan un orden de serialización a priori y ejecutan las transacciones de acuerdo a ellas. Para establecer este ordenamiento, el administrador de transacciones le asigna a cada transacción Ti una estampa de tiempo única ts(Ti) cuando ésta inicia.

Cuando esta inicia una estampa de tiempo es un identificador simple que sirve para identificar cada transacción de manera única. Otra propiedad de las estampas de tiempo es la monoticidad, esto es, dos estampas de tiempo generadas por el mismo administrador de transacciones deben ser monotonicamente crecientes. Así, las estampas de tiempo son valores derivados de un dominio totalmente ordenado.

3

Unidad 3

Programación con acceso a base de datos MySQL



Bases de datos II

Autor: Julio Alberto Castillo

AREANDINA
Fundación Universitaria del Área Andina

MIEMBRO DE LA RED
ILUMNO

Introducción

Esta cartilla está dirigida a realizar un proceso de empalme y continuidad con respecto al módulo de Bases de Datos II, teniendo en cuenta que en ella se ven conceptos básicos de esta temática, el propósito es profundizar y ampliar la cantidad de instrucciones, conceptos y métodos que los estudiantes pueden adquirir para administrar, diseñar y manejar Bases de Datos relacionales.

Todo este proceso será realizado de manera incremental, teniendo como apoyo el desarrollo y aplicación de ejemplos que lleven al estudiante a entender y manejar de manera hábil dichas instrucciones, además que genera su propio estilo de manejo y administración, obviamente sin apartarse de los estándares y métodos que las diferentes entidades han implementado para su desarrollo.

Cada una de las instrucciones que explicaremos en este módulo será tendiente a resolver una problemática propuesta desde el inicio con el fin de llevar una secuencialidad en la forma de resolver las diferentes inquietudes y además generar coherencia dentro de los resultados que se pretenden obtener por parte de los estudiantes.

U3

Metodología

Con el fin de que el estudiante realice la mayor aprensión del conocimiento se realizaran las siguientes recomendaciones metodológicas:

Realizar las lecturas complementarias las cuales le permitirán ampliar conceptos y comprender la temática tratada en la unidad.

Utilizar fuentes bibliográficas e información de internet, recolectada para una mayor comprensión de la información sobre los temas propuestos.

Clasificar la información recolectada y realizar modelos aplicativos en los cuales el estudiante pueda corroborar y experimentar el funcionamiento y las diferentes maneras que hay para trabajar las múltiples funcionalidades que tiene el lenguaje SQL.

U3

Desarrollo temático

Programación con acceso a base de datos en MySQL

Programación con acceso a base de datos

Entre los modelos lógicos, el modelo relacional está considerado como el más simple. No vamos a entrar en ese tema, puesto que es el único que vamos a ver, no tiene sentido establecer comparaciones.

Diremos que es el que más nos conviene. Por una parte, el paso del modelo E-R al relacional es muy simple, y por otra, **MySQL**, como implementación de SQL, está orientado principalmente a bases de datos relacionales.

El modelo se compone de tres partes:

1. Estructura de datos: básicamente se compone de relaciones.
2. Manipulación de datos: un conjunto de operadores para recuperar, derivar o modificar los datos almacenados.
3. Integridad de datos: una colección de reglas que definen la consistencia de la base de datos.

Definiciones

Es importante antes de comenzar esta cartilla, que realicemos la aclaración de terminología que iremos usando de manera paulatina en el desarrollo de la misma, debido a que existen muchas definiciones y algunas de ellas no se acomodan a la realidad estableceremos las que se manejan de manera estándar para los diferentes procesos.

Relación

Es el concepto básico del modelo relacional. Vamos a usar el término interrelación para referirnos a la conexión entre entidades. Para el caso se refiere a una tabla.

Tupla

A menudo se le llama también registro o fila, físicamente es cada una de las líneas de la relación. Equivale al concepto de entidad del modelo E-R, y define un objeto real, ya sea abstracto, concreto o imaginario.

De esta definición se deduce que no pueden existir dos tuplas iguales en la misma relación.

Atributo

También denominado campo o columna, se establece como las divisiones verticales de la relación. Corresponde al concepto de atributo del modelo E-R y contiene cada una de las características que definen una entidad u objeto.

Al igual que en el modelo E-R, cada atributo tiene asignado un nombre y un dominio.

El conjunto de todos los atributos es lo que define a una entidad completa, y es lo que compone una tupla.

Nulo

Hay ciertos atributos, para determinadas entidades, que carecen de valor. El modelo relacional distingue entre valores vacíos y valores nulos. Un valor vacío se considera un valor tanto como cualquiera no vacío, sin embargo, un nulo NULL indica la ausencia de valor.

(NULL) valor asignado a un atributo que indica que no contiene ninguno de los valores del dominio de dicho atributo.

El nulo es muy importante en el modelo relacional, ya que nos permite trabajar con datos desconocidos o ausentes.

Por ejemplo, si tenemos una relación de vehículos en la que podemos almacenar tanto motocicletas como automóviles, un atributo que indique a qué lado está el volante (para distinguir vehículos con el volante a la izquierda de los que lo tienen a la derecha), carece de sentido en motocicletas. En ese caso, ese atributo para entidades de tipo motocicleta será NULL.

Esto es muy interesante, ya que el dominio de este atributo es (derecha, izquierda), de modo que si queremos asignar un valor del dominio no hay otra opción. El valor nulo nos dice que ese atributo no tiene ninguno de los valores posibles del dominio. Así que, en cierto modo amplia la información.

Otro ejemplo, en una relación de personas tenemos un atributo para la fecha de nacimiento. Todas las personas de la relación han nacido, pero en un determinado momento puede ser necesario insertar una para la que desconocemos ese dato. Cualquier valor del dominio será, en principio, incorrecto. Pero tampoco será posible distinguirlo de los valores correctos, ya que será una fecha. Podemos usar el valor NULL para indicar que la fecha de nacimiento es desconocida.

Dominio

El concepto de dominio es el mismo en el modelo E-R y en el modelo relacional. Pero en este modelo tiene mayor importancia, ya que será un dato importante a la hora de dimensionar la relación.

De nuevo estamos ante un concepto muy flexible. Por ejemplo, si definimos un atributo del tipo entero, el dominio más amplio sería, lógicamente, el de los números enteros. Pero este dominio es infinito, y sabemos que los ordenadores no pueden manejar infinitos números enteros. Al definir un atributo de una relación dispondremos de distintas opciones para guardar datos enteros. Si en nuestro caso usamos la variante de "entero pequeño", el dominio estará entre -128 y 127. Pero además, el atributo corresponderá a una característica concreta de una entidad; si se tratase, por ejemplo, de una calificación sobre 100, el dominio estaría restringido a los valores entre 0 y 100.

Modelo relacional

Ahora ya disponemos de los conceptos básicos para definir en qué consiste el modelo relacional. Es un modelo basado en relaciones, en la que cada una de ellas cumple determinadas condiciones mínimas de diseño:

- No deben existir dos tuplas iguales.
- Cada atributo sólo puede tomar un único valor del dominio, es decir, no pueden contener listas de valores.

- El orden de las tuplas dentro de la relación y el de los atributos, dentro de cada tupla, no es importante.

Cardinalidad

La cardinalidad puede cambiar, y de hecho lo hace frecuentemente, a lo largo del tiempo: siempre se pueden añadir y eliminar tuplas.

Grado

El grado de una relación es un valor constante. Esto no quiere decir que no se puedan agregar o eliminar atributos de una relación; lo que significa es que si se hace, la relación cambia. Cambiar el grado, generalmente, implicará modificaciones en las aplicaciones que hagan uso de la base de datos, ya que cambiarán conceptos como claves e interrelaciones, de hecho, puede cambiar toda la estructura de la base de datos.

Esquema: es la parte constante de una relación, es decir, su estructura.

Instancia: es el conjunto de las tuplas que contiene una relación en un momento determinado.

Es como una fotografía de la relación, que sólo es válida durante un periodo de tiempo concreto.

Clave: es un conjunto de atributos que identifica de forma unívoca a una tupla. Puede estar compuesto por un único atributo o una combinación de varios.

Dentro del modelo relacional no existe el concepto de clave múltiple. Cada clave sólo puede hacer referencia a una tupla de una tabla. Por lo tanto, todas las claves de una relación son únicas.

Podemos clasificar las claves en distintos tipos:

- Candidata: cada una de las posibles claves de una relación, en toda relación existirá al menos una clave candidata. Esto implica que ninguna relación puede contener tuplas repetidas.
- Primaria: (o principal) es la clave candidata elegida por el usuario para identificar las tuplas. No existe la necesidad, desde el punto de vista de la teoría de bases de datos relacionales, de elegir una clave primaria. Además, las claves primarias no pueden tomar valores nulos. Es preferible, por motivos de optimización de MySQL, que estos valores sean enteros, aunque no es obligatorio. MySQL sólo admite una clave primaria por tabla, lo cual es lógico, ya que la definición implica que sólo puede existir una.
- Alternativa: cada una de las claves candidatas que no son clave primaria, si es que existen.
- Foránea: (o externa) es el atributo (o conjunto de atributos) dentro de una relación que contienen claves primarias de otra relación. No hay nada que impida que ambas relaciones sean la misma.

Interrelación

Decimos que dos relaciones están interrelacionadas cuando una posee una clave foránea de la otra. Cada una de las claves foráneas de una relación establece una interrelación con la relación donde esa clave es la principal.

Según esto, existen dos tipos de interrelación:

- La interrelación entre entidades fuertes y débiles.

- La interrelación pura, entre entidades fuertes.

Estrictamente hablando, sólo la segunda es una interrelación, pero como veremos más tarde, en el modelo relacional ambas tienen la forma de relaciones, al igual que las entidades compuestas, que son interrelaciones con atributos añadidos.

Al igual que en el modelo E-R, existen varios tipos de interrelación:

- Uno a uno: a cada tupla de una relación le corresponde una y sólo una tupla de otra.
- Uno a varios: a cada tupla una relación le corresponden varias en otra.
- Varios a varios: cuando varias tuplas de una relación se pueden corresponder con varias tuplas en otra.

Triggers

Un disparador es un objeto con nombre en una base de datos que se asocia con una tabla, y se activa cuando ocurre un evento en particular para esa tabla.

```
CREATE TRIGGER nombre_disp momento_disp evento_disp
    ON nombre_tabla FOR EACH ROW sentencia_disp
```

El disparador queda asociado a la tabla *nombre_tabla*. Esta debe ser una tabla permanente, no puede ser una tabla TEMPORARY ni una vista.

momento_disp es el momento en que el disparador entra en acción. Puede ser BEFORE (antes) o AFTER (después), para indicar que el disparador se ejecute antes o después que la sentencia que lo activa.

evento_disp indica la clase de sentencia que activa al disparador. Puede ser INSERT, UPDATE, o DELETE. Por ejemplo, un disparador BEFORE para sentencias INSERT podría utilizarse para validar los valores a insertar.

No puede haber dos disparadores en una misma tabla que correspondan al mismo momento y sentencia. Por ejemplo, no se pueden tener dos disparadores BEFORE UPDATE. Pero sí es posible tener los disparadores BEFORE UPDATE y BEFORE INSERT o BEFORE UPDATE y AFTER UPDATE.

sentencia_disp es la sentencia que se ejecuta cuando se activa el disparador. Si se desean ejecutar múltiples sentencias, deben colocarse entre BEGIN ... END, el constructor de sentencias compuestas. Esto además posibilita emplear las mismas sentencias permitidas en procedimientos almacenados.

Ejemplo: a continuación presentaremos un modelo de cómo se pueden escribir disparadores como el llamado avanzaref, que se muestra en este ejemplo:

```
CREATE TABLE avanza1(a1 INT);
CREATE TABLE avanza2(a2 INT);
CREATE TABLE avanza3(a3 INT NOT NULL AUTO_INCREMENT PRIMARY KEY);
CREATE TABLE avanza4(a4 INT NOT NULL AUTO_INCREMENT PRIMARY KEY, b4 INT DEFAULT 0);

DELIMITER |

CREATE TRIGGER avanzaref BEFORE INSERT ON avanza1
  FOR EACH ROW BEGIN
    INSERT INTO avanza2 SET a2 = NEW.a1;
    DELETE FROM avanza3 WHERE a3 = NEW.a1;
    UPDATE avanza4 SET b4 = b4 + 1 WHERE a4 = NEW.a1;
  END
|

DELIMITER ;
```

```
INSERT INTO avanza3 (a3) VALUES (NULL), (NULL), (NULL), (NULL), (NULL), (NULL), (NULL), (NULL), (NULL), (NULL);

INSERT INTO avanza4 (a4) VALUES 0), (0), (0), (0), (0), (0), (0), (0), (0), (0);

INSERT INTO avanza1 VALUES (1), (3), (1), (7), (1), (8), (4), (4);
```

Imagen 1. Modelo General de un Trigger
Fuente: Propia.

Las columnas de la tabla asociada con el disparador pueden referenciarse empleando los alias OLD y NEW. OLD.nombrecol hace referencia a una columna de una fila existente, antes de ser actualizada o borrada. NEW.nombrecol hace referencia a una columna en una nueva fila a punto de ser insertada, o en una fila existente luego de que fue actualizada.

El uso de SET NEW.nombrecol = valor necesita que se tenga el privilegio UPDATE sobre la columna. El uso de SET nombre_var = NEW.nombrecol necesita el privilegio SELECT sobre la columna.

Nota: actualmente, los disparadores no son activados por acciones llevadas a cabo en cascada por las restricciones de claves extranjeras. Esta limitación se subsanará tan pronto como sea posible.

La sentencia CREATE TRIGGER necesita el privilegio SUPER.

Sintaxis de DROP TRIGGER

```
DROP TRIGGER [nombre_esquema.]nombre_disp
```

Elimina un disparador. El nombre de esquema es opcional. Si el esquema se omite, el disparador se elimina en el esquema actual.

La sentencia DROP TRIGGER necesita que se posea el privilegio SUPER.

Utilización de disparadores

El soporte para disparadores es básico, por lo tanto hay ciertas limitaciones en lo que puede hacerse con ellos. Esta sección trata sobre el uso de los disparadores y las limitaciones vigentes.

Un disparador es un objeto de base de datos con nombre que se asocia a una tabla, y se activa cuando ocurre un evento en particular para la tabla. Algunos usos para los disparadores es verificar valores a ser insertados o llevar a cabo cálculos sobre valores involucrados en una actualización.

Un disparador se asocia con una tabla y se define para que se active al ocurrir una sentencia INSERT, DELETE, o UPDATE sobre dicha tabla. Puede también establecerse que se active antes o después de la sentencia en cuestión. Por ejemplo, se puede tener un disparador que se active antes de que un registro sea borrado, o después de que sea actualizado.

Para crear o eliminar un disparador, se emplean las sentencias CREATE TRIGGER y DROP TRIGGER. La sintaxis de las mismas se describe en.

Este es un ejemplo sencillo que asocia un disparador con una tabla para cuando reciba sentencias INSERT. Actúa como un acumulador que suma los valores insertados en una de las columnas de la tabla.

La siguiente sentencia crea la tabla y un disparador asociado a ella:

```
CREATE TABLE cuentas (num_cuenta INT, monto DECIMALcuenta10,2));

CREATE TRIGGER insertasuma BEFORE INSERT ON cuentas

  FOR EACH ROW SET @sum = @sum + NEW.monto;
```

La sentencia CREATE TRIGGER crea un disparador llamado insertasuma que se asocia con la tabla cuentas. También se incluyen cláusulas que especifican el momento de activación, el evento activador, y qué hacer luego de la activación:

- La palabra clave BEFORE indica el momento de acción del disparador. En este caso, el disparador debería activarse antes de que cada registro se inserte en la tabla. La otra palabra clave posible aquí es AFTER.

- La palabra clave INSERT indica el evento que activará al disparador. En el ejemplo, la sentencia INSERT causará la activación. También pueden crearse disparadores para sentencias DELETE y UPDATE.
- La sentencia siguiente, FOR EACH ROW, define lo que se ejecutará cada vez que el disparador se active, lo cual ocurre una vez por cada fila afectada por la sentencia activadora. En el ejemplo, la sentencia activada es un sencillo SET que acumula los valores insertados en la columna monto. La sentencia se refiere a la columna como NEW.monto, lo que significa "el valor de la columna monto que será insertado en el nuevo registro".

Para utilizar el disparador, se debe establecer el valor de la variable acumulador a cero, ejecutar una sentencia INSERT, y ver qué valor presenta luego la variable.

```
SET @sum = 0;

INSERT INTO cuentas VALUES(137,14.98),(141,1937.50),(97,-100.00);

SELECT @sum AS 'Total monto insertado';
```

En este caso, el valor de @sum luego de haber ejecutado la sentencia INSERT es 14.98 + 1937.50 - 100, o 1852.48.

Para eliminar el disparador, se emplea una sentencia DROP TRIGGER. El nombre del disparador debe incluir el nombre de la tabla:

```
DROP TRIGGER cuentas.insertasuma;
```

Debido a que un disparador está asociado con una tabla en particular, no se pueden tener múltiples disparadores con el mismo nombre dentro de una tabla. También se debería tener en cuenta que el espacio de nombres de los disparadores puede cambiar en el futuro de un nivel de tabla a un nivel de base de datos, es decir, los nombres de disparadores ya no sólo deberían ser únicos para cada tabla sino para toda la base de datos. Para una mejor compatibilidad con desarrollos futuros, se debe intentar emplear nombres de disparadores que no se repitan dentro de la base de datos.

Adicionalmente al requisito de nombres únicos de disparador en cada tabla, hay otras limitaciones en los tipos de disparadores que pueden crearse. En particular, no se pueden tener dos disparadores para una misma tabla que sean activados en el mismo momento y por el mismo evento. Por ejemplo, no se pueden definir dos BEFORE INSERT o dos AFTER UPDATE en una misma tabla. Es improbable que esta sea una gran limitación, porque es posible definir un disparador que ejecute múltiples sentencias empleando el constructor de sentencias compuestas BEGIN ... END luego de FOR EACH ROW. (Más adelante en esta sección puede verse un ejemplo).

También hay limitaciones sobre lo que puede aparecer dentro de la sentencia que el disparador ejecutará al activarse:

- El disparador no puede referirse a tablas directamente por su nombre, incluyendo la misma tabla a la que está asociado. Sin embargo, se pueden emplear las palabras clave OLD y NEW. OLD se refiere a un registro existente que va a borrarse o que va a actualizarse antes de que esto ocurra. NEW se refiere a un registro nuevo que se insertará o a un registro modificado luego de que ocurre la modificación.
- El disparador no puede invocar procedimientos almacenados utilizando la sentencia CALL. (Esto significa, por ejemplo, que no se puede utilizar un procedimiento almacenado para eludir la prohibición de referirse a tablas por su nombre).
- El disparador no puede utilizar sentencias que inicien o finalicen una transacción, tal como START TRANSACTION, COMMIT, o ROLLBACK.

Las palabras clave OLD y NEW permiten acceder a columnas en los registros afectados por un disparador. (OLD y NEW no son sensibles a mayúsculas). En un disparador para INSERT, solamente puede utilizarse NEW. nomcol; ya que no hay una versión anterior del registro. En un disparador para DELETE sólo puede emplearse OLD.nomcol, porque no hay un nuevo registro. En un disparador para UPDATE se puede emplear OLD.nomcol para referirse a las columnas de un registro antes de que sea actualizado, y NEW. nomcol para referirse a las columnas del registro luego de actualizarlo.

Una columna precedida por OLD es de sólo lectura. Es posible hacer referencia a ella pero no modificarla. Una columna precedida por NEW puede ser referenciada si se tiene el privilegio SELECT sobre ella. En un disparador BEFORE, también es posible cambiar su valor con SET NEW.nombrecol = valor si se tiene el privilegio de UPDATE sobre ella. Esto significa que un disparador puede usarse para modificar los valores antes que se inserten en un nuevo registro o se empleen para actualizar uno existente.

En un disparador BEFORE, el valor de NEW para una columna AUTO_INCREMENT es 0, no el número secuencial que se generará en forma automática cuando el registro sea realmente insertado.

OLD y NEW son extensiones de MySQL para los disparadores.

Empleando el constructor BEGIN... END, se puede definir un disparador que ejecute sentencias múltiples. Dentro del bloque BEGIN, también pueden utilizarse otras sintaxis permitidas en rutinas almacenadas, tales como condicionales y bucles. Como sucede con las rutinas almacenadas, cuando se crea un disparador que ejecuta sentencias múltiples, se hace necesario redefinir el delimitador de sentencias si se ingresará el disparador a través del programa MySQL, de forma que se pueda utilizar el carácter ‹;› dentro de la definición del disparador. El siguiente ejemplo ilustra estos aspectos. En él se crea un disparador para UPDATE, que verifica los valores utilizados para actualizar cada columna, y modifica el valor para que se encuentre en un rango de 0 a 100. Esto debe hacerse en un disparador BEFORE porque los valores deben verificarse antes de emplearse para actualizar el registro:

```
delimiter //
CREATE TRIGGER upd_check BEFORE UPDATE ON account
  FOR EACH ROW
  BEGIN
    IF NEW.amount < 0 THEN
      SET NEW.amount = 0;
    ELSEIF NEW.amount > 100 THEN
      SET NEW.amount = 100;
    END IF;
  END;//
delimiter ;
```

Podría parecer más fácil definir una rutina almacenada e invocarla desde el disparador utilizando una simple sentencia CALL. Esto sería ventajoso también si se deseara invocar la misma rutina desde distintos disparadores. Sin embargo, una limitación de los disparadores es que no pueden utilizar CALL. Se debe escribir la sentencia compuesta en cada CREATE TRIGGER donde se la desee emplear.

MySQL gestiona los errores ocurridos durante la ejecución de disparadores de esta manera:

- Si lo que falla es un disparador BEFORE, no se ejecuta la operación en el correspondiente registro.
- Un disparador AFTER se ejecuta solamente si el disparador BEFORE (de existir) y la operación se ejecutaron exitosamente.
- Un error durante la ejecución de un disparador BEFORE o AFTER deriva en la falla de toda la sentencia que provocó la invocación del disparador.
- En tablas transaccionales, la falla de un disparador (y por lo tanto de toda la sentencia) debería causar la cancelación (rollback) de todos los cambios realizados por esa sentencia. En tablas no transaccionales, cualquier cambio realizado antes del error no se ve afectado.

SQL Embebido

Es una de las principales características que se maneja actualmente con SQL. Hay dos razones principales por las que podríamos querer utilizar SQL desde un lenguaje nativo:

- Hay consultas que no se pueden formular utilizando SQL puro (por ejemplo, las consultas recursivas). Para ser capaz de realizar esas consultas necesitamos un lenguaje nativo de mayor poder expresivo que SQL.
- La segunda razón es cuando queremos acceder a una base de datos desde una aplicación que está escrita en el lenguaje nativo elegido (C, Cobol, Java etc.) vale la pena aclarar que se debe usar con una base de datos que puede accederse utilizando SQL embebido, no todas pueden trabarse bajo este esquema.

Un programa que utiliza SQL embebido en un lenguaje nativo consiste en instrucciones del lenguaje e instrucciones de SQL embebido (ESQL). Cada instrucción de ESQL em-

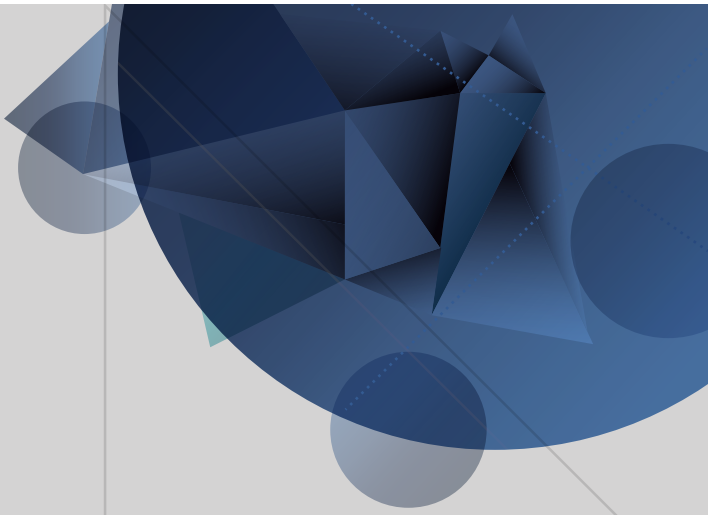
pieza con las palabras claves EXEC SQL. Las instrucciones ESQL se transforman en instrucciones del lenguaje del host mediante un pre compilador (que habitualmente inserta llamadas a rutinas de librerías que ejecutan los variados comandos de SQL).

Cuando vemos los ejemplos de Select observamos que el resultado de las consultas es algo muy próximo a un conjunto de tuplas. La mayoría de los lenguajes no están diseñados para operar con conjuntos, de modo que necesitamos un mecanismo para acceder a cada tupla única del conjunto de tuplas devueltas por una instrucción SELECT. Este mecanismo puede ser proporcionado declarando un cursor. Tras ello, podemos utilizar el comando FETCH para recuperar una tupla y apuntar el cursor hacia la siguiente tupla.

4

Unidad 4

Optimización



Bases de datos II

Autor: Julio Alberto Castillo

AREANDINA
Fundación Universitaria del Área Andina

MIEMBRO DE LA RED
ILUMNO

Introducción

Esta cartilla está dirigida a realizar un proceso de empalme y continuidad con respecto al Módulo de Bases de Datos II, teniendo en cuenta que en ella se ven conceptos básicos de esta temática, el propósito es profundizar y ampliar la cantidad de instrucciones, conceptos y métodos que los estudiantes pueden adquirir para administrar, diseñar y manejar Bases de Datos relacionales.

Todo este proceso será realizado de manera incremental, teniendo como apoyo el desarrollo y aplicación de ejemplos que lleven al estudiante a entender y manejar de manera hábil dichas instrucciones, además que genera su propio estilo de manejo y administración, obviamente sin apartarse de los estándares y métodos que las diferentes entidades han implementado para su desarrollo.

Cada una de las instrucciones que explicaremos en este módulo será tendiente a resolver una problemática propuesta desde el inicio con el fin de llevar una secuencialidad en la forma de resolver las diferentes inquietudes y además generar coherencia dentro de los resultados que se pretenden obtener por parte de los estudiantes.

U4

Metodología

Con el fin de que el estudiante realice la mayor aprensión del conocimiento se realizaran las siguientes recomendaciones metodológicas:

Realizar las lecturas complementarias las cuales le permitirán ampliar conceptos y comprender la temática tratada en la unidad.

Utilizar fuentes bibliográficas e información de internet, recolectada para una mayor comprensión de la información sobre los temas propuestos.

Clasificar la información recolectada y realizar modelos aplicativos en los cuales el estudiante pueda corroborar y experimentar el funcionamiento y las diferentes maneras que hay para trabajar las múltiples funcionalidades que tiene el lenguaje SQL.

Optimización

Manejo de procedimientos almacenados y cursores

Los procedimientos almacenados y funciones son funcionalidades de MySQL. Un procedimiento es un conjunto de comandos SQL que pueden almacenarse en el servidor. Una vez que se hace, los usuarios no necesitan volver a escribir el código ni los comandos individuales, pero pueden en su lugar hacer referencia al procedimiento almacenado.

Los procedimientos almacenados pueden ser particularmente útiles:

- Cuando múltiples aplicaciones cliente se escriben en distintos lenguajes o funcionan en distintas plataformas, pero necesitan realizar la misma operación en la base de datos.
- Cuando la seguridad es un factor muy importante. Las entidades financieras, por ejemplo, usan procedimientos almacenados para todas las operaciones comunes. Esto proporciona un entorno seguro y consistente, y los procedimientos pueden asegurar que cada operación se registra apropiadamente. En este tipo de entorno, las aplicaciones y los usuarios no tendrán acceso directo a las tablas de la base de datos, sólo pueden ejecutar algunos procedimientos almacenados.

Los procedimientos almacenados pueden mejorar el rendimiento ya que hay menos tráfico entre el servidor y el cliente. El intercambio que hay es que aumenta la carga del servidor de la base de datos ya que la mayoría del trabajo se realiza en la parte del servidor y no en el cliente.

Los procedimientos almacenados le permiten tener bibliotecas o funciones en el servidor de base de datos. Esta característica es compartida por los lenguajes de programación modernos que permiten este diseño interno, por ejemplo, el uso de clases. Usando estas características del lenguaje de programación cliente se genera un beneficio para el programador incluso fuera del entorno de la base de datos.

Procedimientos almacenados y las tablas de permisos

El manejo de permisos para la ejecución de diferentes instrucciones se ha modificado para tener en cuenta los procedimientos almacenados como sigue:

- El permiso CREATE ROUTINE se necesita para crear procedimientos almacenados.
- El permiso ALTER ROUTINE se necesita para alterar o borrar procedimientos almacenados. Este permiso se da automáticamente al creador de una rutina.
- El permiso EXECUTE se requiere para ejecutar procedimientos almacenados. Sin embargo, este permiso se da automáticamente al creador de la rutina. También, la característica SQL SECURITY por defecto para una rutina es DEFINER, lo que permite a los usuarios que tienen acceso a la base de datos ejecutar la rutina asociada.

Sintaxis de procedimientos almacenados

Los procedimientos almacenados y rutinas se crean con comandos CREATE PROCEDURE y CREATE FUNCTION. Una rutina es un procedimiento o una función. Un procedimiento se invoca usando un comando CALL, y sólo puede pasar valores usando variables de salida. Una función puede llamarse desde dentro de un comando como cualquier otra función simplemente, invocando el nombre de la función y puede retornar un valor escalar. Las rutinas almacenadas pueden llamar otras rutinas almacenadas.

Los procedimientos almacenados o funciones se asocian con una base de datos. Esto tiene varias implicaciones:

- Cuando se invoca la rutina, se realiza implícitamente USE nombre_bd (y se deshace cuando acaba la rutina). Recuerda que el comando USE no se permite dentro de procedimientos almacenados.
- Cuando se borra una base de datos, todos los procedimientos almacenados asociados con ella también se borran.

Los procedimientos almacenados son globales y no asociados con una base de datos. Heredan la base de datos por defecto del llamador. Si se ejecuta USE nombre_bd desde la rutina, la base de datos por defecto original se restaura a la salida de la rutina.

A continuación se describe la sintaxis usada para crear, modificar, borrar, y consultar procedimientos almacenados y funciones.

CREATE PROCEDURE y CREATE FUNCTION

```
CREATE PROCEDURE nombre_procedimiento ([parametro[,...]])
   [caracteristicas.] cuerpo_rutina
CREATE FUNCTION nombre_funcion ([parametro[,...]])
   RETURNS type
   [caracteristicas..] cuerpo_rutina
parametros:
   [ IN | OUT | INOUT ] nombre_parametro type
tipo:
   Any valid MySQL data type
caracteristicas:
   LANGUAGE SQL
 | [NOT] DETERMINISTIC
 | { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }
 | SQL SECURITY { DEFINER | INVOKER }
 | COMMENT 'string'
Cuerpo_rutina:
```

Para crear una rutina, es necesario tener el permiso CREATE ROUTINE, y los permisos ALTER ROUTINE y EXECUTE se asignan automáticamente a su creador.

Por defecto, la rutina se asocia con la base de datos que está en uso. Para asociar la rutina explícitamente con una base de datos, especifique el nombre como nombre_db.nombre_rutina al crearlo.

La cláusula RETURNS puede especificarse sólo con FUNCTION, donde es obligatorio. Se usa para indicar el tipo de dato que retorna la función, y el cuerpo de la función debe contener un comando RETURN value.

La lista de parámetros entre paréntesis debe estar siempre presente. Si no hay parámetros, se debe usar una lista de parámetros vacía (). Cada parámetro es un parámetro IN por defecto. Para especificar otro tipo de parámetro, use la palabra clave OUT o INOUT antes del nombre del parámetro. Especificando IN, OUT, o INOUT sólo es válido para una PROCEDURE.

Un marco para procedimientos almacenados externos se introducirá próximamente. Esto permitirá escribir procedimientos almacenados en lenguajes distintos a SQL.

Para replicación, use la función NOW() (o su sinónimo) o RAND() no hace una rutina no determinista necesariamente. Para NOW(), el log binario incluye el tiempo y hora y replica correctamente. RAND() también replica correctamente mientras se invoque sólo una vez dentro de una rutina.

Varias características proporcionan información sobre la naturaleza de los datos usados por la rutina. CONTAINS SQL indica que la rutina no contiene comandos que leen o escriben datos. NO SQL indica que la rutina no contiene comandos SQL. READS SQL DATA indica que la rutina contiene comandos que leen datos, pero no comandos que escriben datos. MODIFIES SQL DATA indica que la rutina contiene comandos que pueden escribir datos. CONTAINS SQL es el valor por defecto si no se dan explícitamente ninguna de estas características.

MySQL permite a las rutinas que contengan comandos DDL (tales como CREATE y DROP) y comandos de transacción SQL (como COMMIT). Los procedimientos almacenados no pueden usar LOAD DATA INFILE.

Los comandos que retornan un conjunto de resultados no pueden usarse desde una función almacenada. Esto incluye comandos SELECT que no usan INTO para tratar valores de columnas en variables, comandos SHOW y otros comandos como EXPLAIN. Para comandos que pueden determinarse al definir la función para que retornen un conjunto de resultados, aparece un mensaje de error Not allowed to return a result set from a function (ER_SP_NO_RETSET_IN_FUNC).

El siguiente es un ejemplo de un procedimiento almacenado que use un parámetro OUT. El ejemplo usa el cliente mysql y el comando delimiter para cambiar el delimitador del comando de; a // mientras se define el procedimiento. Esto permite pasar el delimitador; usado en el cuerpo del procedimiento a través del servidor en lugar de ser interpretado por el mismo mysql.

```
delimiter //
CREATE PROCEDURE nombre ([parámetro1, parámetro2,...])
[Atributos de la rutina]
BEGIN instrucciones
END
   //
delimiter ;
```

Al usar el comando delimiter, debe evitar el uso de la antibarra ('\') ya que es el carácter de escape de MySQL.

El siguiente es un ejemplo de función que toma un parámetro, realiza una operación con una función SQL, y retorna el resultado:

```
delimiter //
CREATE FUNCTION trabajo (s CHAR(20)) RETURNS CHAR(50)
     RETURN CONCAT('Prueba , ',s ,'!');
     //
delimiter ;
SELECT trabajo('resultado');
```

Imagen 1. Creación Función
Fuente: Propia.

Si el comando RETURN en un procedimiento almacenado retorna un valor con un tipo distinto al especificado en la cláusula RETURNS de la función, el valor de retorno se convierte al tipo apropiado. Por ejemplo, si una función retorna un valor ENUM o SET, pero el comando RETURN retorna un entero, el valor retornado por la función es la cadena para el miembro de ENUM correspondiente de un conjunto de miembros SET.

ALTER PROCEDURE y ALTER FUNCTION

```
ALTER {PROCEDURE | FUNCTION} nombre_proc [caracteristicas ...]
caracteristicas:
  { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }
 | SQL SECURITY { DEFINER | INVOKER }
 | COMMENT 'string'
```

Este comando puede usarse para cambiar las características de un procedimiento o función almacenada. Debe tener el permiso ALTER ROUTINE para la rutina desde MySQL. El permiso se otorga automáticamente al creador de la rutina.

Pueden especificarse varios cambios con ALTER PROCEDURE o ALTER FUNCTION.

DROP PROCEDURE y DROP FUNCTION

```
DROP {PROCEDURE | FUNCTION} [IF EXISTS] nombre_proc
```

Este comando se usa para borrar un procedimiento o función almacenada. Esto borra la rutina especificada del servidor. Debe tener el permiso ALTER ROUTINE. Este permiso se otorga automáticamente al creador de la rutina.

La cláusula IF EXISTS es una extensión de MySQL. Evita que ocurra un error si la función o procedimiento no existe. Se genera una advertencia que puede verse con SHOW WARNINGS.

SHOW CREATE PROCEDURE y SHOW CREATE FUNCTION

```
SHOW CREATE {PROCEDURE | FUNCTION} nombre_proc
```

Este comando es similar a SHOW CREATE TABLE, retorna la cadena exacta que puede usarse para recrear la rutina nombrada.

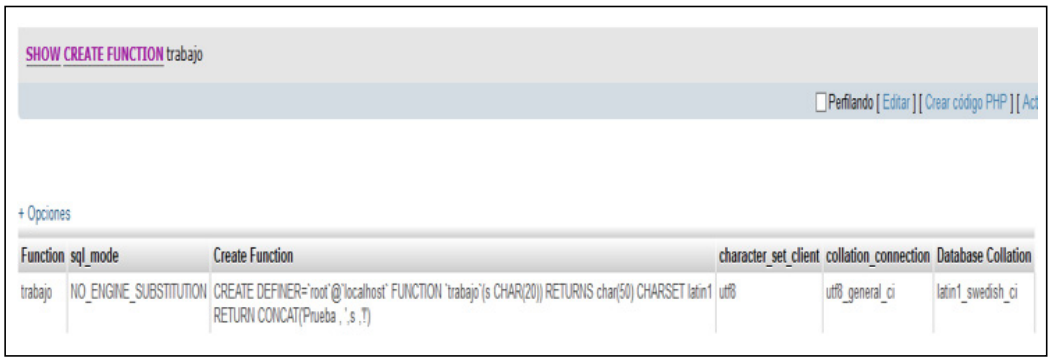


Imagen 2. Mostrar Estructura con Show
Fuente: Propia.

SHOW PROCEDURE STATUS y SHOW FUNCTION STATUS

```
SHOW {PROCEDURE | FUNCTION} STATUS [LIKE 'pattern']
```

Este comando retorna características de las rutinas, como el nombre de la base de datos, nombre, tipo, creador y fechas de creación y modificación. Si no se especifica un patrón, le lista la información para todos los procedimientos almacenados, en función del comando que use.

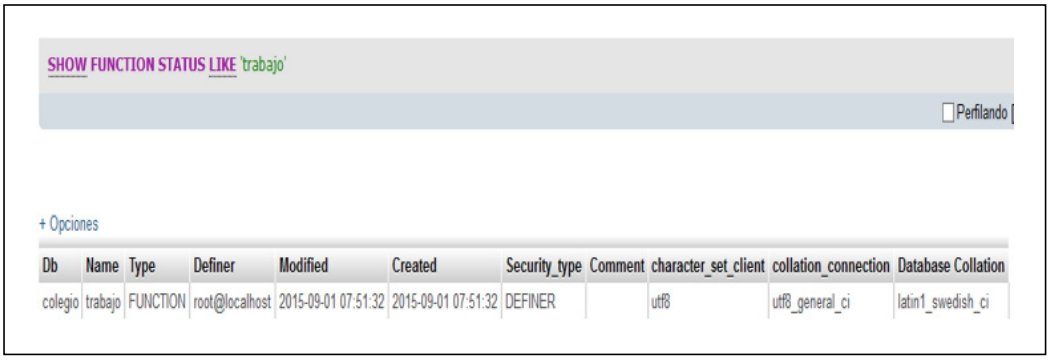


Imagen 3. Ejecución Show Status
Fuente: Propia.

La sentencia CALL

```
CALL sp_name([parameter[,...]])
```

El comando CALL invoca un procedimiento definido previamente con CREATE PROCEDURE.

CALL puede pasar valores al usuario que la invoca usando parámetros declarados como OUT o INOUT.

Sentencia compuesta BEGIN ... END

```
[etiqueta_inicio:] BEGIN
  [lista_sentencias]
END [etiqueta_fin]
```

La sintaxis BEGIN... END se utiliza para escribir sentencias compuestas que pueden aparecer en el interior de procedimientos almacenados y triggers. Una sentencia compuesta puede contener múltiples sentencias, encerradas por las palabras BEGIN y END. lista_sentencias es una lista de una o más sentencias. Cada sentencia dentro de lista_sentencias debe terminar con un punto y como (;) delimitador de sentencias. lista_sentencias es opcional, lo que significa que la sentencia compuesta vacía (BEGIN END) es correcta.

El uso de múltiples sentencias requiere que el cliente pueda enviar cadenas de sentencias que contengan el delimitador; Esto se gestiona en el cliente de línea de comandos mysql con el comando delimiter. Cambiar el delimitador de fin de sentencia; (por ejemplo con //)

permite utilizar; en el cuerpo de una rutina Un comando compuesto puede etiquetarse. No se puede poner end_label a no ser que también esté presente begin_label, si ambos están, deben ser iguales.

Sentencia DECLARE

El comando DECLARE se usa para definir varios iconos locales de una rutina: las variables locales Los comandos SIGNAL y RESIGNAL no se soportan en la actualidad.

DECLARE puede usarse sólo dentro de comandos compuestos BEGIN ... END y deben ser su inicio, antes de cualquier otro comando.

Los cursores deben declararse antes de declarar los handlers, y las variables y condiciones deben declararse antes de declarar los cursores o handlers.

Variables en procedimientos almacenados

Declarar variables locales con DECLARE

Sentencia SET para variables

La sentencia SELECT ... INTO puede declarar y usar una variable dentro de una rutina.

Declarar variables locales con DECLARE

```
DECLARE nombre_var[,...] type [DEFAULT value]
```

Este comando se usa para declarar variables locales. Para proporcionar un valor por defecto para la variable, incluya una cláusula DEFAULT. El valor puede especificarse como expresión, no necesita ser una constante. Si la cláusula DEFAULT no está presente, el valor inicial es NULL.

La visibilidad de una variable local es dentro del bloque BEGIN... END donde está declarado. Puede usarse en bloques anidados excepto aquéllos que declaren una variable con el mismo nombre.

Sentencia SET para variables

```
SET nombre_var = expr [, nombre_var = expr] ...
```

El comando SET en procedimientos almacenados es una versión extendida del comando general SET. Las variables referenciadas pueden ser las declaradas dentro de una rutina, o variables de servidor globales.

El comando SET en procedimientos almacenados se implementa como parte de la sintaxis SET pre-existente. Esto permite una sintaxis extendida de SET a=x, b=y, ... donde distintos tipos de variables (variables declaradas local y globalmente y variables de sesión del servidor) pueden mezclarse. Esto permite combinaciones de variables locales y algunas opciones que tienen sentido sólo para variables de sistema; en tal caso, las opciones se reconocen pero se ignoran.

La sentencia SELECT ... INTO

```
SELECT nombre_columna[,...] INTO nombre_var[,...] tabla_expr
```

Esta instrucción SELECT almacena columnas seleccionadas directamente en variables. Por lo tanto, sólo un registro puede retornarse.

```
SELECT id, dato  INTO x, y FROM test.t1 LIMIT 1;
```

Conditions and Handlers

Condiciones DECLARE

DECLARE handlers

Ciertas condiciones pueden requerir un tratamiento específico. Estas condiciones pueden estar relacionadas con errores, así como control de flujo general dentro de una rutina.

Condiciones DECLARE

```
DECLARE nombre_condicion CONDITION FOR valor_condicion
Valor_condicion:
  SQLSTATE [VALUE] sqlstate_value
 | mysql_error_code
```

Este comando especifica condiciones que necesitan tratamiento específico. Asocia un nombre con una condición de error específica. El nombre puede usarse subsecuentemente en un comando DECLARE HANDLER. Además de valores SQLSTATE, los códigos de error MySQL se soportan.

```
DECLARE handlers
DECLARE tipo_handler HANDLER FOR valor_condicion[,...] sp_statement
tipo_handler:
    CONTINUE
  | EXIT
  | UNDO
valor_condicion:
    SQLSTATE [VALUE] sqlstate_value
  | nombre_condicion
  | SQLWARNING
  | NOT FOUND
  | SQLEXCEPTION
  | mysql_error_code
```

Este comando especifica handlers que pueden tratar una o varias condiciones. Si una de estas condiciones ocurre, el comando especificado se ejecuta.

Para un handler CONTINUE, continúa la rutina actual tras la ejecución del comando del handler. Para un handler EXIT, termina la ejecución del comando compuesto BEGIN...END actual. El handler de tipo UNDO todavía no se soporta.

- SQLWARNING es una abreviación para todos los códigos SQLSTATE que comienzan con 01.
- NOT FOUND es una abreviación para todos los códigos SQLSTATE que comienzan con 02.
- SQLEXCEPTION es una abreviación para todos los códigos SQLSTATE no tratados por SQLWARNING o NOT FOUND.

Además de los valores SQLSTATE , los códigos de error MySQL se soportan.

Por ejemplo:

```
1  CREATE TABLE test.ejemplo (s1 int,primary key (s1));
2
3  delimiter //
4  CREATE PROCEDURE handlerprueba ()
5       BEGIN
6       DECLARE CONTINUE HANDLER FOR SQLSTATE '23000' SET @x2 = 1;
7        SET @x = 1;
8        INSERT INTO test.ejemplo VALUES (1);
9        SET @x = 2;
10       INSERT INTO test.ejemplo VALUES (1);
11       SET @x = 3;
12     END;
13     //
14 CALL handlerprueba()//
15
```

Imagen 4. Manejo de los Handlers en un procedimiento
Fuente: Propia.

Tenga en cuenta que @x es 3, lo que muestra que se ha ejecutado al final del procedimiento. Si la línea DECLARE CONTINUE HANDLER FOR SQLSTATE '23000' SET @x2 = 1; no está presente, MySQL habría tomado la ruta por defecto (EXIT) tras el segundo INSERT fallido debido a la restricción PRIMARY KEY, y SELECT @x habría retornado 2.

Cursores

Se soportan cursores simples dentro de procedimientos y funciones almacenadas. La sintaxis es la de SQL empotrado. Los cursores no son sensibles, son de sólo lectura, y no permiten scrolling. No sensible significa que el servidor puede o no hacer una copia de su tabla de resultados.

Los cursores deben declararse antes de declarar los handlers, y las variables y condiciones deben declararse antes de declarar cursores o handlers.

Por ejemplo:

```
1  CREATE PROCEDURE curprueba()
2  BEGIN
3    DECLARE done INT DEFAULT 0;
4    DECLARE a CHAR(16);
5    DECLARE b, c INT;
6    DECLARE cursor1 CURSOR FOR SELECT id, data FROM test.t1;
7    DECLARE cursor2 CURSOR FOR SELECT i FROM test.t2;
8    DECLARE CONTINUE HANDLER FOR SQLSTATE '02000' SET done = 1;
9
10   OPEN cursor1;
11   OPEN cursor2;
12
13   REPEAT
14     FETCH cursor1 INTO a, b;
15     FETCH cursor2 INTO c;
16     IF NOT done THEN
17        IF b < c THEN
18
19           INSERT INTO test.t3 VALUES (a, b);
20        ELSE
21           INSERT INTO test.t3 VALUES (a, c);
22        END IF;
23     END IF;
24   UNTIL done END REPEAT;
25
26   CLOSE cursor1;
27   CLOSE cursor2;
28 END
29
```

Imagen 5. Ejemplo de un Cursor
Fuente: Propia.

Declarar cursores

```
DECLARE nombre_cursor CURSOR FOR select_statement
```

Este comando declara un cursor. Pueden definirse varios cursores en una rutina, pero cada cursor en un bloque debe tener un nombre único.

El comando SELECT no puede tener una cláusula INTO.

Sentencia OPEN del cursor

```
OPEN nombre_cursor
```

Este comando abre un cursor declarado previamente.

Sentencia de cursor FETCH

```
FETCH nombre_cursor INTO nom_var [, nom_var] ...
```

Este comando trata el siguiente registro (si existe) usando el cursor abierto especificado, y avanza el puntero del cursor.

Sentencia de cursor CLOSE

```
CLOSE nombre_cursor
```

Este comando cierra un cursor abierto previamente.

Si no se cierra explícitamente, un cursor se cierra al final del comando compuesto en que se declara.

Constructores de control de flujo

Sentencia IF

```
IF search_condition THEN statement_list
   [ELSEIF search_condition THEN statement_list] ...
   [ELSE statement_list]
END IF
```

IF implementa un constructor condicional básico. Si search_condition se evalúa a cierto, el comando SQL correspondiente listado se ejectua. Si no coincide ninguna search_condition se ejecuta el comando listado en la cláusula ELSE. statement_list puede consistir en varios comandos.

Tenga en cuenta que también hay una función IF(), que difiere del commando IF descrito aquí.

La sentencia CASE

```
CASE case_value
  WHEN when_value THEN statement_list
  [WHEN when_value THEN statement_list] ...
  [ELSE statement_list]
END CASE
O:
CASE
  WHEN search_condition THEN statement_list
  [WHEN search_condition THEN statement_list] ...
  [ELSE statement_list]
END CASE
```

El comando CASE para procedimientos almacenados implementa un constructor condicional complejo. Si una search_condition se evalúa a cierto, el comando SQL correspondiente se ejecuta. Si no coincide ninguna condición de búsqueda, el comando en la cláusula ELSE se ejecuta.

Nota: La sintaxis de un comando CASE mostrado aquí para uso dentro de procedimientos almacenados difiere ligeramente de la expresión CASE SQL. El comando CASE no puede tener una cláusula ELSE NULL y termina con END CASE en lugar de END.

Sentencia LOOP

```
[begin_label:] LOOP
  statement_list
END LOOP [end_label]
```

LOOP implementa un constructor de bucle simple que permite ejecución repetida de comandos particulares. El comando dentro del bucle se repite hasta que acaba el bucle, usualmente con un comando LEAVE

Un comando LOOP puede etiquetarse. end_label no puede darse hasta que esté presente begin_label , y si ambos lo están, deben ser el mismo.

Sentencia LEAVE

```
LEAVE label
```

Este comando se usa para abandonar cualquier control de flujo etiquetado. Puede usarse con BEGIN ... END o bucles.

La sentencia ITERATE

```
ITERATE label
```

ITERATE sólo puede aparecer en comandos LOOP, REPEAT y WHILE. ITERATE significa "vuelve a hacer el bucle."

Por ejemplo:

```
1  CREATE PROCEDURE doiterate(p1 INT)
2  BEGIN
3    label1: LOOP
4      SET p1 = p1 + 1;
5      IF p1 < 10 THEN ITERATE label1; END IF;
6      LEAVE label1;
7    END LOOP label1;
8    SET @x = p1;
9  END
```

Imagen 6. Crear procedimiento
Fuente: Propia.

Sentencia REPEAT

```
[begin_label:] REPEAT
  statement_list
UNTIL search_condition
END REPEAT [end_label]
```

El comando/s dentro de un comando REPEAT se repite hasta que la condición search_condition es cierta.

Un comando REPEAT puede etiquetarse. end_label no puede darse a no ser que begin_label esté presente, y si lo están, deben ser el mismo.

Por ejemplo:

```
delimiter //

CREATE PROCEDURE dorepeat(p1 INT)
     BEGIN
       SET @x = 0;
       REPEAT SET @x = @x + 1; UNTIL @x > p1 END REPEAT;
     END
     //
CALL dorepeat(1000)//

SELECT @x//
```

Imagen 7. Ejemplo repetición
Fuente: Propia.

Sentencia WHILE

```
[begin_label:] WHILE search_condition DO
  statement_list
END WHILE [end_label]
```

El comado dentro de un comando WHILE se repite mientras la condición search_condition es cierta.

Un comando WHILE puede etiquetarse. end_label no puede darse a no ser que begin_label también esté presente, y si lo están, deben ser el mismo.

Por ejemplo:

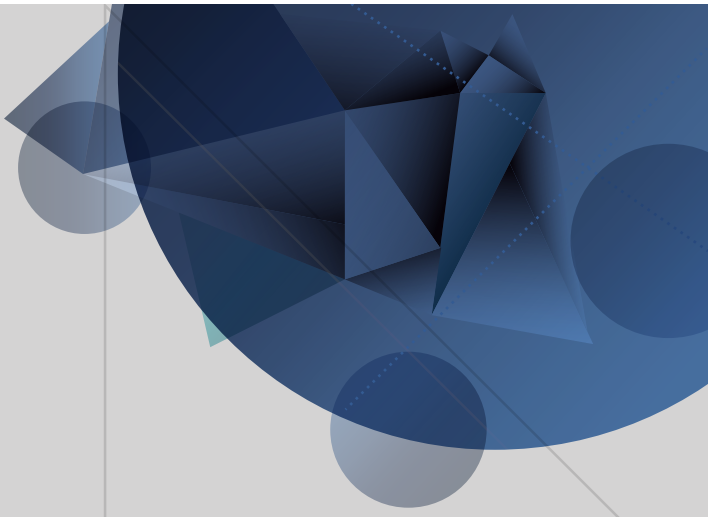
```
CREATE PROCEDURE dowhile()
BEGIN
  DECLARE v1 INT DEFAULT 5;

  WHILE v1 > 0 DO
    ...
    SET v1 = v1 - 1;
  END WHILE;
END
```

Imagen 8. Ejemplo While
Fuente: Propia.

4

Unidad 4

Las bases de Datos no relacionales NoSQL



Bases de datos II

Autor: Julio Alberto Castillo

AREANDINA
Fundación Universitaria del Área Andina

MIEMBRO DE LA RED
ILUMNO

Introducción

Esta cartilla está dirigida a realizar un proceso de empalme y continuidad con respecto al módulo de Bases de Datos II, teniendo en cuenta que en ella se ven conceptos básicos de esta temática, el propósito es profundizar y ampliar la cantidad de instrucciones, conceptos y métodos que los estudiantes pueden adquirir para administrar, diseñar y manejar Bases de Datos relacionales.

Todo este proceso será realizado de manera incremental, teniendo como apoyo el desarrollo y aplicación de ejemplos que lleven al estudiante a entender y manejar de manera hábil dichas instrucciones, además que genera su propio estilo de manejo y administración, obviamente sin apartarse de los estándares y métodos que las diferentes entidades han implementado para su desarrollo.

Cada una de las instrucciones que explicaremos en este módulo será tendiente a resolver una problemática propuesta desde el inicio con el fin de llevar una secuencialidad en la forma de resolver las diferentes inquietudes y además generar coherencia dentro de los resultados que se pretenden obtener por parte de los estudiantes.

U4

Metodología

Con el fin de que el estudiante realice la mayor aprensión del conocimiento se realizaran las siguientes recomendaciones metodológicas:

Realizar las lecturas complementarias las cuales le permitirán ampliar conceptos y comprender la temática tratada en la unidad.

Utilizar fuentes bibliográficas e información de internet, recolectada para una mayor comprensión de la información sobre los temas propuestos.

Clasificar la información recolectada y realizar modelos aplicativos en los cuales el estudiante pueda corroborar y experimentar el funcionamiento y las diferentes maneras que hay para trabajar las múltiples funcionalidades que tiene el lenguaje SQL.

Las Bases de Datos no relacionales NoSQL

NoSQL visto como el futuro de las bases de datos, pros y contras

Actualmente las bases de datos ya no se entienden como hace algunos años: estas han evolucionado con el auge del Internet y los motores de bases de datos no se han quedado atrás.

El panorama que tenemos las personas es vivir súper conectados, esto ha derivado en la generación de millones de datos por segundo a través de aplicaciones online, redes sociales, blogs, foros, entre otros.

La lluvia de información trae múltiples soluciones, pero de la misma manera también problemas a nuestras vidas, pero estos problemas son mucho mayores a las personas encargadas de mantener la información de una manera integral para las personas, por lo tanto entender las bases de datos como un almacén de información es un grave error en el que podemos estar incurriendo ahora mismo.

El mundo conectado

Hace 20 años era fácil vivir sin un teléfono celular, sin una cuenta de correo electrónico, sin acceder a Internet. Hoy en día no contar con uno o todos estos medios es impensable por la misma dinámica del sistema globalizado: la familia, amigos, las empresas, las universidades todos ellos usan redes sociales; estas últimas utilizan sistemas de e-learning, muchas empresas utilizan sistema de gestión para manejar sus procesos, las entidades financieras apoyan el comercio electrónico, mercados y muchas otras actividades que demuestran que la dinámica está cambiando hacia tener acceso a todos los servicios a través de Internet, incluso está planteado y ya es una realidad, medicina a distancia y objetos conectados entre objetos (The Internet of Things).

Esta avalancha de conexiones tiene como efecto colateral la generación de datos que son almacenados en grandes repositorios de datos de los proveedores de servicios, los ISPs.

La situación actual de los sistemas de bases de datos

Existen dos fuertes tendencias en el universo de las bases de datos: uno es el movimiento "SQL" y otro el movimiento "NoSQL", ambos con sus pros y contras.

Las bases de datos SQL se les llama bases de datos relacionales, y son de las que más se tiene conocimiento, por ser modelo de estudio en las carreras universitarias relacionadas con sistemas e informática, han sido de mucha utilidad para hacer aplicaciones

transaccionales en las cuales mantener y proteger la integridad de la información.

En la otra esquina tenemos las "NoSQL", acrónimo de Not Only SQL, que apunta más por la evolución del Internet; no es exactamente un tipo de base de datos, sino un conjunto de tipos de bases de datos, por ejemplo, con las bases de datos documentales que son las más conocidas, se podría hacer prácticamente todo lo que se hace actualmente con una relacional. Cabe destacar que en NoSQL no existe ACID (Atomicidad, coherencia, aislamiento, durabilidad) como en las bases de datos relacionales, pero existe algo llamado BASE (disponibilidad básica, estado suave, consistencia eventual) que da características para el manejo de datos.

En la mayoría de los sistemas se usan bases de datos relacionales, por ser las más tradicionales y las que más se conocen. Muchos de los grandes sitios como Facebook, Twitter, Google y Yahoo se han cimentado en MySQL o PostgreSQL, pero en los últimos años se han dado cuenta de que esto no es suficiente y se originó la llamada "Guerra de las bases de datos".

Algunos de los protagonistas de esta guerra son organizaciones muy conocidas: Facebook, Apache, LinkedIn. Aunque muchos servicios empiezan a usar NoSQL hay otros que son dependientes de SQL: el cambio implicaría un costo enorme. Una de esas empresas es Twitter: emplea tanto SQL como NoSQL en sus sistemas, con lo que se demuestra que ambas pueden convivir perfectamente.

SQL como el pasado de las bases de datos

Las bases de datos relacionales son muy utilizadas actualmente, pero empiezan a ser insuficientes respecto a la forma de almacenamiento de información en Internet, con lo que se generan cuellos de botella y estructuras muy rígidas que impiden un crecimiento constante, a la par con las fuentes de datos.

Hay personas tratando de mantener la llama encendida, como el proyecto MariaDB que ha introducido el motor de Cassandra (una base de datos NoSQL) demostrando que muchas personas apuestan por la interoperabilidad de SQL y NoSQL.

NoSQL como el futuro

Aunque el NoSQL se ve como el futuro de los sistemas de bases de datos, hay movimientos importantes: el afán de Facebook por crear motores de bases de datos como Cassandra y RocksDB, y el uso de NoSQL en otras redes sociales como Twitter y Linkedin, que indican que todo apunta hacia servicios que usen repositorios datos con NoSQL únicamente.

Pero hay dos eventos que impiden esta evolución. El primero es la resistencia al cambio pues se piensa que los motores de bases de datos no pueden resolver muchos problemas de la vida real (aunque la verdad los proyectos no usan ni el 50% de las ventajas que ofrece SQL) teniendo como argumento que no se garantiza la integridad de los datos, o la consistencia.

Muchas personas siguen viendo el tema de NoSQL apreciado desde un punto de vista SQL: es como comparar un Renault 4 con un Ferrari, ambos sirven para transportarse, pero el funcionamiento de cada uno de ellos no es similar, manejar un Renault 4 pensando que es un Ferrari es un grave error.

El otro problema es la diversidad de tipos de NoSQL que existen: clave/valor, documentos, grafos, tabular, entre otras. Las personas se confunden y abruman al tener que decidir cuál usar. Esa elección puede ser errada. Siempre será una apuesta muy fuerte saber si tu proyecto se alineará adecuadamente con el tipo de base de datos que se ha elegido.

Para hacer una elección efectiva es recomendable tener claro lo siguiente:

1. Qué tipo de proyecto es, si es para Internet o intranet.
2. Cuál es la cantidad estimada de usuarios y si muchas de las funciones deben hacer uso de las transacciones complejas, o si, de lo contrario son escrituras y lecturas manejadas desde el software.
3. Qué tecnología se va a usar para el desarrollo (si usa Django por ejemplo, lo recomendable es usar BD Relacional, ya que es más natural el manejo de los datos en aplicaciones hechas en ese framework).

Con esos tres puntos se puede ver las opciones, qué base de datos se adapta más a lo que se busca.

Cambiando la manera de pensar

Existe un mito, muy difundido entre los seguidores acérrimos de SQL, y es que piensan y divulgan que "en bases de datos NoSQL no existen estructuras". Pero no es cierto, en NoSQL existen estructuras, aunque hay que aclarar que son más flexibles en comparación a SQL y de aquí se deriva el término schema-free.

Las bases de datos convencionales son de tipo relacional, es decir, constan de "relaciones de datos", que se corresponden entre sí para almacenar de la forma más eficiente la información. Se consultan estos datos usando SQL (Structured Query Language), un lenguaje estructurado para manejar la información y muy popular por cierto.

La generación de nuevos tipos de aplicaciones urgidos de gestionar enormes cantidades de datos, que deben ser introducidos y seleccionados rápidamente y además que deben poder crecer de manera sencilla y barata. En estos sistemas importa más la flexibilidad, la velocidad y la capacidad de escalado horizontal que otras cuestiones tradicionalmente cruciales como la consistencia o disponer de una estructura perfectamente definida para los datos. Es más, las tradicionales características ACID de las bases de datos relacionales no son una premisa para los nuevos gestores de información.

Por todo ello existe una tendencia imparable hacia la definición y uso de un nuevo tipo de bases de datos de tipo documental, que en lugar de registrar relaciones de datos almacenan documentos con estructuras arbitrarias y cumplen con las premisas expuestas anteriormente. Sus principales características definitorias son, por un lado, la enorme flexibilidad que brindan en los esquemas de datos a la hora de almacenar la información y su simplicidad de uso, pero también su altísimo rendimiento y escalabilidad, la facilidad de mantenimiento y la capacidad de funcionar sin puntos de fallo pues pueden recuperarse aunque se caiga cualquiera de sus nodos.

Además, la forma de consultarlas, extraer y resumir información es completamente distinta a lo que estamos acostumbrados con SQL, por eso a este tipo de datos documentales se les denomina también bases de

datos no-relacionales, documentales o más comúnmente Bases de datos NoSQL (de Not Only SQL).

Se suele confundir el concepto NoSQL con el de base de datos documental. En realidad éstas últimas son una categoría concreta dentro de las bases de datos NoSQL que también incluyen almacenes de pares clave-valor (como Dynamo o MotionDb), servidores de estructuras de datos (como Redis), almacenes de N-Tuplas (como Apache River) o almacenes de filas (como Cassandra o BigTable, desarrollada por Google), entre otros tipos. Conviene no confundirlas, más adelante conoceremos un poco más de algunas de ellas.

Escalabilidad

La escalabilidad de un sistema se define como la capacidad que se tiene para expandirse según las necesidades que se deriven del uso que se le va a dar. De este modo, por ejemplo, una aplicación web es escalable si es posible atender a un número cada vez mayor de usuarios sin necesidad de cambiar la aplicación.

Fundamentalmente existen dos maneras de escalar:

- Escalado vertical: este implica agregar más recursos al sistema actual para que pueda atender cada vez más solicitudes. Así se le añaden más procesadores o memoria a un servidor o se añade más espacio de almacenamiento. Este sistema de escalado tiene un límite y es que llega un punto en el que la aplicación necesita ser rediseñada para poder escalar más.
- Escalado horizontal: por otro lado este escalado se consigue simplemente añadiendo más nodos al sistema, por ejemplo, colocando otra máquina en paralelo a funcionar. Este tipo de escalado es más sencillo y no tiene límite, pero implica que las aplicaciones estén adaptadas para poder hacerlo, pues implican coordinación, replicación, etc.

Los sistemas NoSQL son sistemas de almacenamiento de información ágil, de gran rendimiento, distribuido y, en casi todos los casos, escalables horizontalmente.

Esto último es muy importante porque permiten manejar eficientemente volúmenes de datos muy grandes y además escalar a voluntad con una gran simplicidad, lo cual los hace muy atractivos para aplicaciones que puedan llegar a crecer mucho. Por ello son muy utilizados por todas las grandes empresas de Internet, juegos, apps masivas para móviles, etc.

El teorema de Brewer o teorema CAP

Eric Brewer, un profesor de la Universidad de Berkeley, formuló un teorema acerca de las tres dimensiones de los sistemas distribuidos (de almacenamiento de datos en este caso), que son:

- Consistencia *(Consistency)*: tiene que ver con la permanente coherencia y consistencia que deben tener los datos después de cualquier operación, de modo que cualquier usuario que acceda a ella verá la misma información.
- Disponibilidad *(Availability)*: toda la información del sistema de almacenamiento de datos distribuido está siempre disponible.
- Tolerancia de las particiones *(Partition Tolerance)*: las diferentes partes del sistema distribuido continuarán funcionando normalmente aunque la comunicación

entre ellos se vea interrumpida parcial o totalmente o no sea confiable.

Según el teorema no es posible cumplir con las tres dimensiones a la vez. Hay que elegir dos de ellas y de las 3 parejas resultantes se generan diferentes tipos de sistemas distribuidos de almacenamiento:

- C y A: son sistemas en los que si falla la comunicación entre sus nodos el conjunto no puede trabajar.
- C y P: en estos, si algo ocurre parte de la información no estará disponible, pero seguirán funcionando y la información disponible será consistente.
- A y P: durante un fallo de uno de los nodos (o falta de comunicación) la información estará disponible pero puede que no sea consistente.

Gracias a estas parejas podemos clasificar los sistemas de almacenamiento de datos y conocer a qué categoría pertenece cada uno de los sistemas más populares, sabiendo en cada caso qué características nos pueden proporcionar y si se adaptan a nuestras necesidades. La mejor forma de verlo es de manera gráfica:

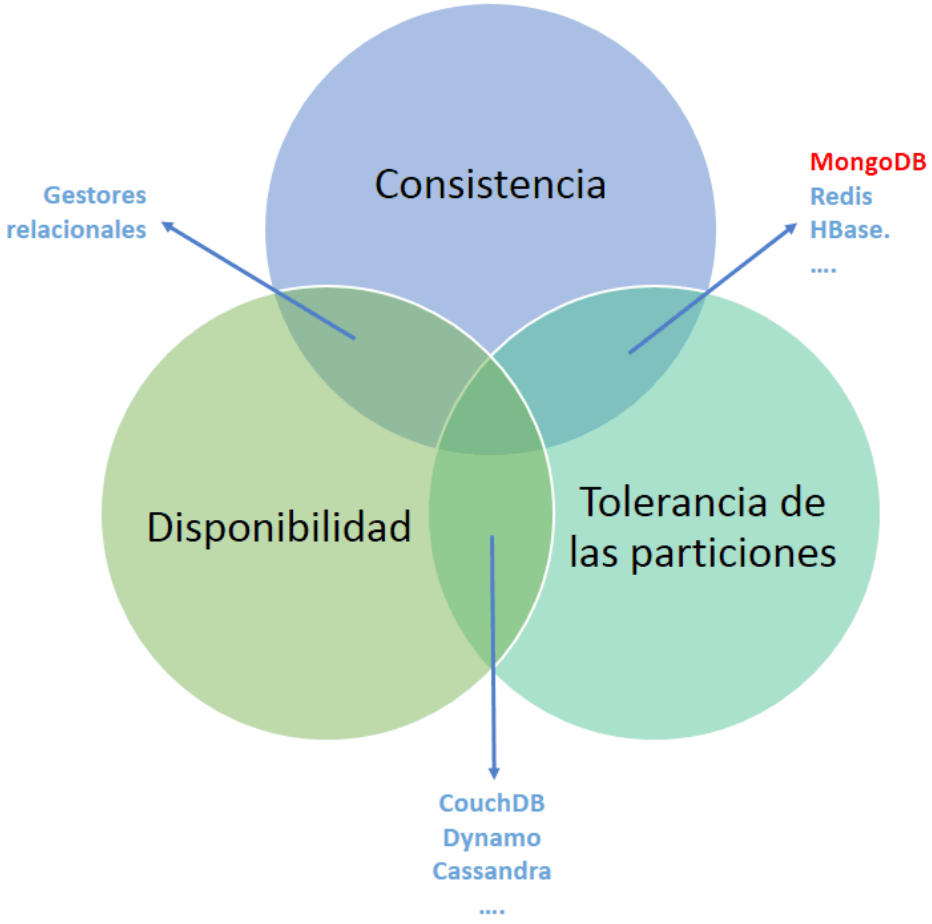


Imagen 1
Fuente: http://www.campusmvp.es/recursos/image.axd?picture=CAP.png

A continuación veremos algunas generalidades de los principales motores con los cuales se nutre el sistema NoSQL, para dar una idea más clara de cuáles son sus características y algunos aspectos generales de su desarrollo y uso.

MongoDB

Es un gestor de datos NoSQL distribuido de tipo documental que almacena documentos en un formato similar a JSON con el cual se programa en muchos de los Frameworks de Javascript (para ser más exactos internamente usa BSON). Está escrita en C++ y es multi-plataforma, Open Source y gratuito.

El proyecto nació a finales del año 2007 como un proyecto interno de una empresa llamada 10Gen para usarlo en una aplicación de Internet que estaban desarrollando, pero en 2009 decidieron liberarlo como Open Source y dedicarse íntegramente a él, ofreciendo soporte comercial y servicios relacionados.

Su nombre proviene de la palabra en inglés Humongous, que significa literalmente "algo realmente grande", y se refiere a su capacidad de gestionar grandes cantidades de datos.

Sus principales características son:

- Basado en el motor V8 de Google Chrome para JavaScript. Facilidad de aprendizaje por basarse en este lenguaje.
- Almacenamiento flexible basado en JSON sin necesidad de definir esquemas previamente.
- Soporte para creación de índices a partir de cualquier atributo, lo que facilita mucho su uso para porque no es necesario definir procesos Map-Reduce.
- Alto rendimiento para consultas y actualizaciones.
- Consultas flexibles basadas en documentos.
- Alta capacidad de crecimiento, replicación y escalabilidad: puedes escalar horizontalmente simplemente añadiendo máquinas baratas sin ver afectado el rendimiento ni complicar la gestión.
- Soporte para almacenamiento independiente de archivos de cualquier tamaño basado en GridFS.
- Soporte para tareas Map-Reduce si es necesario.

Aplicaciones de MongoDB

Este tipo de bases de datos no sustituyen a las bases de datos tradicionales, como SQL Server, Oracle o MySQL, sino que las complementan para ciertos tipos de aplicaciones especializadas. Las bases de datos documentales como MongoDB se utilizan para multitud de tareas, pero fundamentalmente cuando necesitamos flexibilidad en la definición de los datos, sencillez a la hora de acceder a éstos, gran rendimiento y posibilidad de crecer muy rápido.

Es adecuada para crear aplicaciones de Internet que registren muchos datos o que simplemente queramos crear de manera muy flexible, pero también para sistemas muy grandes como registradores de datos de sensores, que pueden llegar a recibir cientos de miles de lecturas de datos por segundo, pasando por gestores de datos de ventas, infraestructura de almacenamiento para redes sociales, juegos masivos online, gestores de contenidos, aplicaciones de análisis de datos y reporteria.

Actualmente lo utilizan para empresas como eBay, Foursquare, SourceForge, The New York Times, The Guardian, SAP, o el propio CERN en su gran colisionador de hadrones, pero también muchas empresas pequeñas que quieren poder desarrollar de manera ágil, barata, sencilla y sin miedo a poder crecer más adelante.

Al estar basada en JavaScript se lleva especialmente bien con todo tipo de aplicaciones web, especialmente las más modernas Single Page Applications.

En general MongoDB puede utilizarse para casi cualquier cosa para la que utilizarías SQL Server o MySQL, pero sin la rigidez que presentan este tipo de bases de datos.

Contendientes en servicios de nube ya hay un número de vendedores de servicios web que ofrecen en la actualidad bases de datos en las cuales el concepto principal es que se "paga según utilizas". Cada uno ofrece características únicas y variaciones respecto a los estándares generales aquí descritos.

Daremos un vistazo a bases de datos particulares, tales como SimpleDB, Google AppEngine, Datastore y SQL Data Services.

Amazon: SimpleDB

Es una base de datos clave/valor orientada a atributos disponible en la plataforma de Amazon Web Services. SimpleDB es todavía una Beta pública. Mientras tanto, los usuarios pueden registrarse para una versión "gratuita" - limitada por uso.

SimpleDB tiene algunas limitaciones. La primera es que una consulta no puede durar más de 5 segundos. La segunda, que no hay más tipos de datos que cadenas de texto. Cada almacenamiento, recuperación o comparación se realiza en formato de cadena de texto (string), por lo que las comparaciones de fechas no funcionan a menos que se éstas se conviertan al formato ISO8601. La tercera, el tamaño máximo de cada cadena de texto es de 1024 caracteres, lo cual limita mucho los textos (como descripciones de productos, etc.) que puedes almacenar un atributo simple. Pero debido a que el esquema es dinámico y flexible, puedes superar esta limitación añadiendo atributos «DescripcionProducto1», «DescripcionProducto2», etc. El límite por cada elemento es de 256 atributos. Mientras SimpleDB sea Beta, los dominios no pueden ser más grandes de 10GB, y las bases de datos enteras no pueden superar 1TB.

Una característica clave de SimpleDB es que utiliza un modelo de consistencia eventual. Este modelo de consistencia es bueno para la concurrencia, ya que después de haber cambiado un atributo en un elemento, los cambios no serán reflejados en las operaciones de lectura inmediatas a dicho cambio. Esto hay que tenerlo muy en cuenta en casos como, por ejemplo, cuando no quieras vender la última entrada de un concierto en el sistema de reserva para cinco personas debido a que los datos no fueron consistentes en tiempo de venta.

Ventajas

Baja interacción

El servicio le permite centrarse completamente en el desarrollo de aplicaciones que añadan valor, en lugar de dedicar mucho tiempo a arduas tareas de administración de bases de datos. Amazon SimpleDB gestiona automáticamente el aprovisionamiento de la infraestructura, el mantenimiento

del hardware y del software, la replicación e indización de elementos de datos y el ajuste del rendimiento.

Alta disponibilidad

Amazon SimpleDB crea automáticamente varias copias de cada elemento almacenado y las distribuye geográficamente. De esta forma se ofrece alta disponibilidad y capacidad de duración: en el caso improbable de que falle una réplica, Amazon SimpleDB puede realizar una conmutación por error utilizando otra réplica del sistema.

Flexibilidad

A medida que cambie su negocio o evolucione la aplicación, podrá reflejar dichos cambios fácilmente en Amazon SimpleDB sin necesidad de preocuparse por si rompe un esquema rígido o necesita refactorizar código – basta con añadir otro atributo a su conjunto de datos en Amazon SimpleDB cuando lo necesite. También puede elegir entre solicitudes de lectura coherentes o finalmente coherentes, que le permiten ajustar los requisitos de rendimiento de lectura (latencia y rendimiento) y de coherencia con las demandas de la aplicación, o incluso diferenciar partes de la aplicación.

Facilidad de uso

Amazon SimpleDB racionaliza el acceso a las funciones de almacenamiento y consultas que suelen llevarse a cabo utilizando un clúster de bases de datos relacionales, al tiempo que excluye otras operaciones de bases de datos que son complejas y que muchas veces ni se utilizan. El servicio permitirá agregar datos con rapidez y recuperar o editar fácilmente datos mediante un sencillo conjunto de llamadas a la API.

Google AppEngine Datastore

Está construido con BigTable, el sistema de almacenamiento interno de Google para la manipulación de datos estructurados. En sí mismo, Google AppEngine Datastore no es un mecanismo de acceso directo a BigTable, sino una especie de interfaz simplificada superpuesta a BigTable.

El AppEngine Store soporta muchos más tipos de datos y elementos que SimpleDB, incluyendo tipos de lista, que contiene colecciones dentro de un elemento simple.

Sin embargo, no podrás realizar interfaces con el AppEngine Datastore (o con BigTable) utilizando aplicaciones externas a la plataforma de servicios web de Google.

Microsoft SQL Data Services

Es parte de la plataforma Azure Web Services de Microsoft. El SDS es un servicio que está en versión Beta y es gratuito con algunas limitaciones en el tamaño de las bases de datos. SQL Data Services es en sí misma una aplicación situada en lo alto de muchos servidores SQL, los cuales crean el almacenamiento de datos por debajo de la plataforma SDS. Mientras que el almacenamiento por debajo pueda ser relacional, tú no tienes acceso a éste; SDS es un almacenamiento clave/valor, como otras plataformas discutidas anteriormente.

Microsoft parece estar sola entre los tres tipos de vendedores en reconocimiento de que mientras el almacenamiento clave/valor es fenomenal para la escalabilidad, acarrea un gran gasto de gestión de datos cuando es comparado con RDBMS. La aproximación de Microsoft parece ser deshacer

hasta los huesos pelados para obtener un buen mecanismo de escalamiento y distribución, y entonces transcurrido el tiempo fortalecerse, añadiendo características que ayuden a puentear los huecos entre el almacenamiento clave/valor y la plataforma de base de datos relacional.

Contendientes en servicios que no son de nube

Fuera de la nube existen vendedores de software de base de datos que pueden ser instalados "en casa". Casi todos estos productos son todavía jóvenes, todavía en versiones alpha o beta, pero en su mayoría son software libre (open source); pudiendo tener acceso al código, tú puedas quizás ser más consciente de cuestiones potenciales y de las limitaciones que tú podrías tener con vendedores de soluciones cerradas.

CouchDB

Es una base de datos orientada a documentos, open source y gratuita. Derivada del almacenamiento clave/valor, utiliza JSON para definir el esquema de un elemento. CouchDB está concebida para tender un puente al hueco que existe entre bases de datos relacionales y bases de datos orientadas a documentos gracias a las "vistas", que pueden ser creadas dinámicamente a través de JavaScript. Estas vistas mapean los datos del documento en estructuras similares a tablas que pueden ser indexadas y consultadas.

El proyecto Voldemort

Es una base de datos clave/valor distribuida que tiene previsto escalar horizontalmente a través de un gran número de servidores. Está engendrado a partir del trabajo realizado en LinkedIn y, según se informa, utilizado allí donde muchos sistemas tengan requerimientos de escalabilidad muy altos. El Proyecto Voldemort también utiliza un modelo de consistencia eventual, basado en el de Amazon.

Drizzle

Puede ser considerado como una contrapropuesta para los problemas que los almacenamientos clave/valor tienen que resolver. La vida de Drizzle comenzó como un resultado indirecto de la base de datos MySQL (6.0). En los últimos meses, sus desarrolladores han removido las características no centrales (sin núcleo) de un host (incluyendo vistas, triggers, sentencias preparadas, procedimientos almacenados, caché de consultas, ACL, y varios tipos de datos), con el objetivo de crear un sistema de base de datos más ligero, simple y rápido. Drizzle puede todavía almacenar datos relacionales. Como apuntó Brian Aker, de MySQL/Sun, "no hay razón de sacar al bebé del agua del baño". El objetivo es construir una plataforma de base de datos semi-relacional a medida para la web y para aplicaciones basadas en la nube ejecutándose en sistemas con 16 nucleos o más.

Tomar una decisión

Últimamente, existen cuatro razones por las cuales deberías optar por una plataforma de base de datos no relacional clave/valor para tus aplicaciones:

1. Tus datos están fuertemente orientados a documentos, haciéndolos más acordes y naturales con el modelo de datos clave/valor que con el modelo de datos relacional.
2. Tu entorno de desarrollo está fuertemente orientado a objetos, y una base de datos clave/valor podría reducir la necesidad de código «tubería».

3. El almacenamiento de datos es económico y se integra fácilmente con la plataforma de servicios web de tu proveedor.
4. El asunto más importante es que es bajo demanda, con alta escalabilidad final -- esto es, gran escala, escalabilidad distribuida, del tipo que no puede ser conseguida simplemente mediante escalar añadiendo.

Para tomar una decisión hay que recordar las limitaciones de las bases de datos y los riesgos que corres por bifurcación con respecto del plan relacional.

Para el resto de requerimientos, tu mejor opción sea una buena y vieja RDBMS. Así pues, ¿están condenadas las bases de datos relacionales? Claramente, no. Al menos, no todavía.

Ha surgido un movimiento denominado NOSQL, con carácter rebelde, criticando las RDBMS y ensalzando las virtudes de las nuevas alternativas algunas compañías que tienen su negocio en Internet, utilizan las bases de datos no relacionales, y ellas mismas han desarrollado su propio sistema, como Google, Microsoft, Amazon o Facebook. Su negocio es inmenso y requieren de miles de servidores repartidos por todo el mundo. Tenemos que ver si éste es nuestro escenario o si, por el contrario el nuestro se reduce a un simple servidor al modo tradicional, o a unos pocos servidores en cluster localizados en el mismo edificio, para lo cual, otras alternativas RDBMS van a ser las más adecuadas, como Oracle, SQL Server, MySQL o PostgreSQL.

Cassandra

Este sistema de base de datos fue desarrollado por Facebook, abandonando el prestigioso MySQL. Este sistema ya forma parte de la incubadora de proyectos de Apache. Cassandra es open source, altamente distribuido, y tiene un modelo de datos similar al de BigTable (Google). Entre sus características cabe destacar la alta disponibilidad, consistencia eventual, escalabilidad incremental, replicación optimista, administración mínima. Posee API para acceder desde lenguajes de programación tales como C++, C#, Java, Perl o PHP y muchos de los Frameworks más populares entre los cuales esta JQuery, Yii, Laravel Etc.

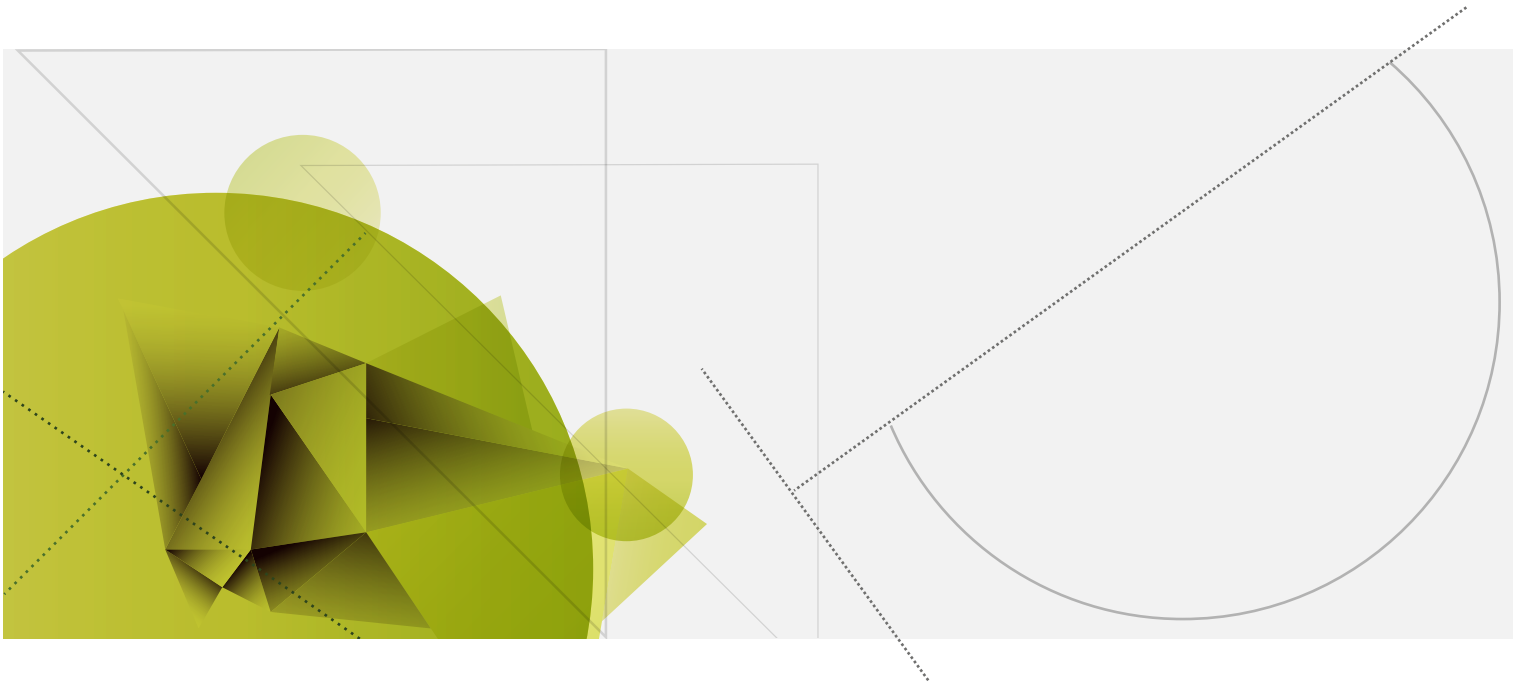
Imagen 2

Fuente http://cdn.blog.celingest.com/wp-content/uploads/2014/10/rdbms_vs_nosql_vs.png

Bibliografia

- Date, C. (2004). An introduction to Database Systems. Ed. Prentice-Hall.
- Date, C. & Darwen, C. (1998) Foundation for Object/Relational Database: The Third Manifesto.
- Elmasri, R. & Navathe, S. (2010). Fundamentals of Database Systems.
- Gray, J. & Reuter, A. (s.f.). Transaction processing: Concepts and Techniques. Morgan Kaufmann Publishers.
- González, C. (2002). Sistemas de Bases de Datos. Tercera reimpresión, Editorial Tecnológica de Costa Rica.
- Groff, J. & Weinberg, P. (2001).A FONDO Microsoft SQL Server 7.0. Editorial Mc GrawHill.
- Han, J. & Kamber, M. (2001). Data Mining: Concepts and Techniques. Morgan Kaufmann Publishers.
- Hansen, G. & Hansen, J. (s.f.) Diseño y Administración de Bases de Datos. Prentice Hall. LAN times Guía de SQL.
- Korth, H. & Silberschatz, A. (2002). Fundamentos de Bases de Datos. McGraw-Hill. Ver Link
- Nguyen, H. & Walker, E. (2001). A First Course in Fuzzy Logic. Second Edition. Chapman & Hall/ CRC.
- Özsu, M. & Valduriez, P. (1999). Principles of Distributed Database Systems. Second Edition. Prentice.
- Ramalho, Y. (2000). SQL Server Iniciación y Referencia. Editorial McGraw-Hill.
- Soukup, R. & Delaney, K. (s.f.) Fundamentos de Bases de datos.
- Ullman, J. & Widom, J. (s.f.). First course in Database Systems.
- Ullman, J. (s.f.). Principles of Database and Knowledge-Base Systems.
- Weikum, G. & Vossen, G. (2000) Transactional Information Systems: Theory, Algorithms, and the practice of Concurrency Control and Recovery. Morgan Kaufmann Publishers.

Esta obra se terminó de editar en el mes de noviembre
Tipografá Myriad Pro 12 puntos
Bogotá D.C,-Colombia.



AREANDINA
Fundación Universitaria del Área Andina

MIEMBRO DE LA RED
ILUMNO